MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un langage de variabilité

Warnon, Philippe

Award date:
2012

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'informatique

Année académique 2011-2012

Implémentation d'un langage de variabilité

Philippe Warnon

Mémoire présenté en vue de l'obtention du grade de master en Sciences Informatiques

Résumé

Depuis plusieurs décennies, les chaines de montages permettent la production de masse de produit personnalisables à des coûts raisonnables. Les « Software Product Lines » (SPLs) sont une adaptation de ces chaines de montages au niveau de l'ingénierie logicielle.

Depuis les années 90, les « Feature Diagrams » sont utilisés afin de modéliser la variabilité des Software Product Lines (SPLs). Le langage TVL permet une représentation textuelle de la variabilité, ce qui offre les avantages d'un langage textuel classique, tel que l'usage d'éditeurs textuels ou d'outils de compilation et de validation.

La première partie de ce mémoire vise à rappeler les différents concepts liés aux SPLs ou aux Feature Diagrams, ainsi que les étapes clés de l'analyse de langage et le langage TVL.

La seconde partie présente des contributions à TVL : l'introduction des cardinalités de feature, qui détermine le nombre de fois qu'un feature et son sous-arbre peuvent être instanciés dans un produit (instance dérivée d'un modèle TVL), et un langage permettant d'exprimer des configurations de produits.

Mots-clés : Software Product Line, SPL, Variabilité, Modélisation de la variabilité, Feature Diagram, Langage textuel de variabilité, TVL, Cardinalités de Feature, Configuration de produit.

Abstract

For several decades, industrial assembly lines have allowed mass production of customized product at reasonable costs. « Software Product Lines » (SPLs) are the adaption of these assembly lines to the software engeneering.

Since the 90's, Feature Diagrams are used to model the variability of the Software product Lines (SPLs). The TVL language allows a textual representation of the variability, which offers the advantages of a classic textual language, such using textual editors or using parsers and validation tools.

The first part of this thesis presents the different concepts of the SPLs, Feature Diagrams, language parsing and the TVL language.

The second part presents extensions of TVL : the introduction of feature cardinalities, which determines the number of times a feature and its subtree can be instanciated in a product (instance derived from a model), and a language allowing to describe configurations of products.

Keywords : Software Product Line, SPL, Varibility, Variability Modelling, Feature Diagram, Textual Variability Language, TVL, Feature Cardinalities, Product Congifuration.

Avant-propos

Je remercie mes promoteurs, le professeur P. Heymans, R. Michel et G. Saval, pour leur suivi, leur disponibilité et leurs conseils tout au long de l'élaboration de ce mémoire.

Je remercie également l'ensemble des professeurs, assistants et membres de la faculté informatique des FUNDP, sans qui je n'aurais pas pu entreprendre l'élaboration de ce mémoire.

Enfin, à titre plus personnel, je remercie Laura et mes proches pour leur soutien, leur patience et leurs encouragements tout au long de mes études.

Table des matières

Glossaire

Voici une liste d'abréviations couramment utilisées dans ce mémoire. Les concepts qu'elles représentent sont définis dans leur contexte dans la suite de ce mémoire.

AST : « Abstract Syntax Tree », structure interne contenant les informations issues de l'analyse syntaxique d'un code source (cfr 3.4).

FD : « Feature Diagram », représentation graphique d'un ensemble d'un feature (cfr 2.1).

LALR : « Look Ahead-Left Recursive », il s'agit d'un des types d'analyse syntaxique qui s'opère de manière ascendante (cfr 3.3.2).

$\mathcal{L}_{\mathbf{exp}}$ **:** Sous-ensenble de TVL permettant d'exprimer des expressions booléennes et arithmétiques (cfr 6.1).

LL : « Left to right, Leftmost Derivation», type d'analyse syntaxique qui s'opère de manière descandante (cfr 3.3.2).

LR : « Left to right, Rightmost dérivation », il s'agit d'un des types d'analyse syntaxique qui s'opère de manière ascendante (cfr 3.3.3).

$\mathcal{L}_{\mathbf{TVL}}$ **:** Langage abstrait pour lequel il existe une correspondance un à un de chaque élément de $\mathrm{TVL}_{\mathbf{NF}}$, la sémantique de ce langage est définie formellement et fournit donc une sémantique au langage TVL (cfr 4.1).

$\mathcal{L}_{\mathbf{TVLP}}$ **:** Langage abstrait pour lequel il existe une correspondance un à un de chaque élément de la version normalisée de TVL-P, la sémantique de ce langage est définie formellement et fournit donc une sémantique au langage TVL-P (cfr 7.7).

p.f.a : L'acronyme « p.f.a » désigne les notations permettant de représenter un ensemble de valeurs issues d'un Set de clones d'un feature dans le cadre de la grammaire permettant de définir les expressions de TVL en y incluant la possibilité du clonage (cfr 6.3.3).

SLR : « Simple Left to right, Rightmost dérivation **»,** il s'agit d'un des types d'analyse syntaxique qui s'opère de manière ascendante (cfr 3.3.2).

SPL : « Software Product Line », ligne de produit logiciel (cfr 1.1).

TVL : « Textual Variability Language », langage textuel permettant de représenter la variabilité des SPL (cfr 4).

$\mathbf{TVL_{NF}}$ **:** Forme normalisée de TVL (cfr 4.1)

TVL-P : Langage permettant d'exprimer la configuration des produits dérivés de modèle exprimés en TVL (cfr 7.2).

Introduction

TVL est l'abréviation de Textual Variability Language, il s'agit donc d'un langage permettant de représenter la variabilité, mais sous forme textuelle.

Ce mémoire est constitué de deux parties, une première va présenter les concepts nécessaires à la compréhension de ce mémoire, la seconde consistera à apporter des contributions à TVL.

Contextualisation

TVL permet d'exprimer la variabilité des Sofware Product Lines (SPLs), un premier chapitre va donc présenter le concept de SPL et introduire le concept de variabilité.

Le second chapitre va présenter les Feature Diagrams. Il s'agit d'une famille de langages sous forme graphique permettant de représenter la variabilité des SPLs.

Mais TVL n'est pas un langage graphique, il s'agit d'un langage sous forme textuelle, avec une syntaxe proche du langage C. Un troisième chapitre rappellera donc les concepts liés à l'analyse des langages.

Après avoir présenté les concepts de variabilité et de langage, le langage de variabilité TVL sera présenté dans un quatrième chapitre.

Contributions

Le cinquième chapitre marque le début des contributions à TVL, il décrira l'introduction du clonage dans TVL : une spécification existe déjà pour cette fonctionnalité, mais est abstraite : elle pourrait être adaptée à différents langages de variabilité. Le but de ce chapitre sera donc de transposer cette spécification abstraite en une spécification propre à TVL et de modifier l'implémentation du parser TVL en conséquence.

Le sixième chapitre proposera une discussion sur un élément particulier du langage TVL : les contraintes. Celles-ci doivent être revues suite à l'introduction du clonage. Toutefois, aucune implémentation de ce point ne sera décrite.

Ensuite, le septième chapitre décrira le langage TVL-P. Il s'agit d'un langage permettant d'exprimer des configurations de produits.

Le huitième chapitre présentera les exigences et la conception d'un outil d'analyse de configuration exprimée dans le langage TVL-P. Cet outil a été implémenté dans le cadre de ce mémoire.

Enfin, une conclusion de ce travail sera présentée.

Partie 1 : Contextualisation

1. Software Product Line

Ce chapitre va présenter brièvement le concept de Software Product Line ou Ligne de produits logiciels (SPL).

L'informatique n'est pas un monde à part, c'est un ensemble d'outils et de techniques, une ingénierie au service de bien d'autres disciplines mais aussi inspirée par ces disciplines.

Les SPLs n'échappent pas à cette règle, elles représentent même un des exemples les plus marquants.

Ce chapitre est principalement inspiré de l'ouvrage « Software Product Line Engineering : Foundations, principles and Techniques» de K Pohl et al. [1].

La définition d'une SPL ainsi que ses deux principales notions, la plateforme et la personnalisation, seront présentées, du point de vue informatique mais aussi avec des analogies dans le monde industriel.

Ensuite, les deux processus intervenant dans le développement des SPLs, le domain engineering et l'application engineering seront présentés.

Enfin, quelques avantages et inconvénients des SPLs seront cités.

1.1 Définition

La littérature nous offre plusieurs définitions, entre autres les suivantes :

L'ingénierie de ligne de produit est, selon Pohl et al. [1], « un paradigme de développement d'applications logicielles utilisant des plateformes et la personnalisation de masse ».

Selon Clements et al. [2], une SPL est « un ensemble de systèmes à forte composante logicielle partageant un ensemble commun et géré de caractéristiques répondant aux besoins spécifiques d'un secteur de marché ou d'une mission et qui sont développés à partir d'une ensemble commun de composants de base d'une manière bien déterminée.

Ces définitions mettent en évidence deux concepts : la **plateforme** (ou ensemble de systèmes à forte composante logicielle) et la **personnalisation** de masse (ou besoin spécifique)….

Ces principes de plateforme et de personnalisation de masse ne sont pas propres à l'informatique, mais sont utilisés couramment dans le monde industriel. Par exemple, dans les chaines de montages automobiles permettant une production de masse tout en offrant bon nombre d'options aux clients afin de personnaliser leur véhicule et augmenter les ventes.

L'industrie nous a prouvé depuis des décennies l'utilité de telles chaines de montages par rapport à la production de produits isolés.

Voici un exemple caractéristique afin d'illustrer différents concepts (issu de Pohl et al.[1]):

Example 1-2: *From the Camera World*

In 1987, Fuji released the Quicksnap, the first single-use camera. It caught Kodak by surprise: Kodak had no such product in a market that grew from then on by 50% annually, from 3 million in 1988 to 43 million in 1994. However, Kodak won back market share and in 1994, it had conquered 70% of the US market. How did Kodak achieve it? First, a series of clearly distinguishable, different camera models was built based on a common platform. Between April 1989 and July 1990, Kodak reconstructed its standard model and created three additional models, all with common components and the same manufacturing process. Thus, Kodak could develop the cameras faster and with lower costs. The different models appealed to different customer groups. Kodak soon had twice as many models as Fuji, conquered shelf space in the shops and finally won significant market share this way (for details see [Robertson and Ulrich 1999; Clark and Wheelwright 1995]).

Dans l'exemple ci-dessus, différents modèles d'appareils photos « Kodak » ont été construits sur base d'une **plateforme** commune.

Ces modèles d'appareils photos font partie d'une même **ligne de produits** (appelée aussi famille de produits).

Ces appareils photos sont les **produits,** ils ont des points communs, par exemple, nous pourrions imaginer que Kodak prévoit le même boitier pour toute une ligne de produit, la même alimentation et le même flash. Ces éléments communs à tous les produits de la ligne font partie de la plateforme.

Mais ces appareils photos peuvent aussi se différencier des autres appareils photos de la ligne, bien qu'ils respectent un socle commun et qu'ils utilisent un maximum de composants communs, on parle alors de **variabilité** (« Variability », le **V** de TVL).

L'informatique s'est inspirée de ces plateformes avec la naissance des SPLs. Une plateforme au sens informatique est définie comme telle :

Definition 1-4: *Software Platform*

A software platform is a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced.

[Meyer and Lehnerd 1997]

Définition issue de Pohl et al., [1], reprise de Meyer and Lehnerd 1997.

La **variabilité** est définie comme étant « the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts» [1].

Afin de mieux comprendre ces concepts de plateforme, points communs et variabilité au niveau informatique, la section suivante présente le processus de développement d'une ligne de produits logiciels.

1.2 Processus de développement

Comme représenté sur la figure ci-dessous, le SPL engineering se compose du Domain Engineering et de l'Application Engineering. Ces deux étapes du processus de développement sont présentées aux points suivants.

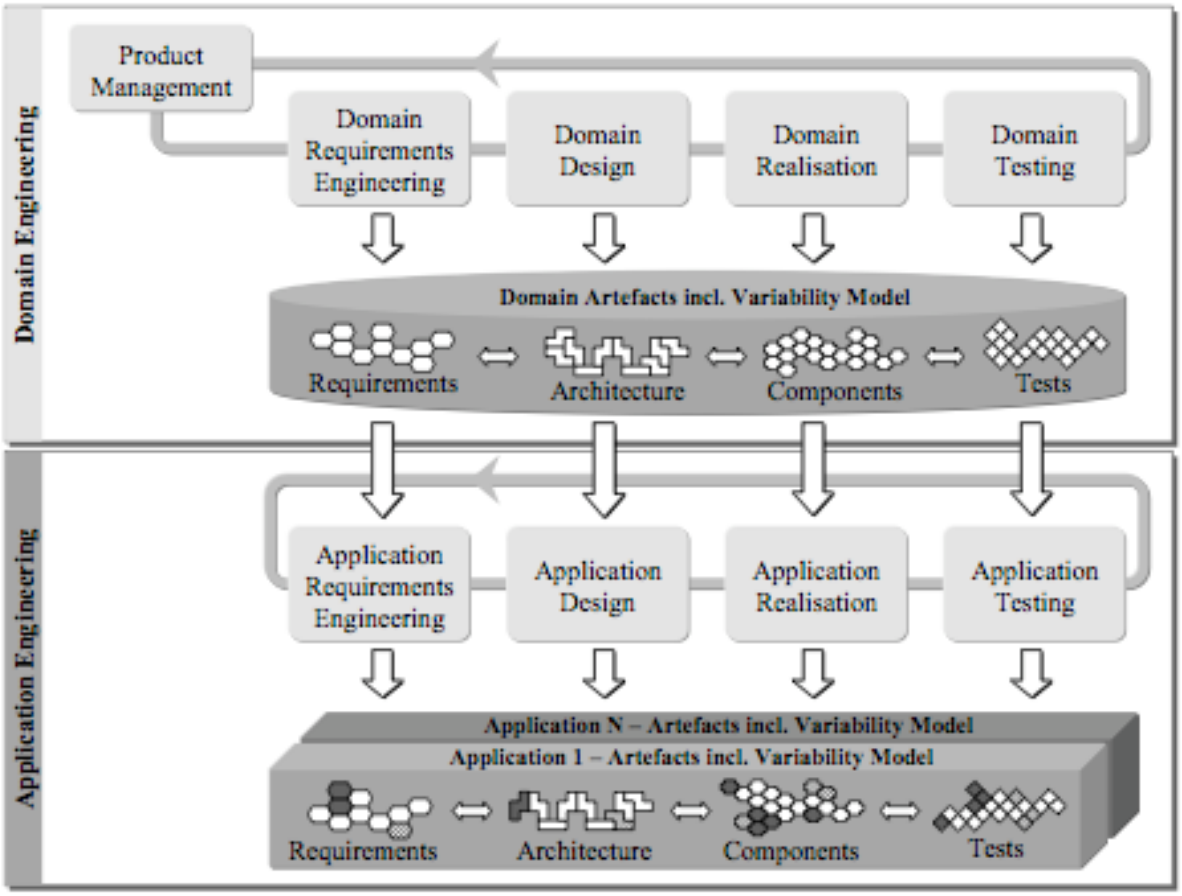


Figure 1.1 **Domain et Application Engineering (Pohl et al [1], page 22).**

1.2.1 Domain Engineering

Ce processus est responsable de la réalisation de la plateforme. Il doit déterminer les points communs des produits appartenant à la ligne de produits ainsi que la flexibilité laissée à chaque produit (**variabilité** du produit).

Il comporte 5 sous-processus (représentés sur la figure 1.1) :

- **Product Management**
  Regroupe les tâches relevant de la gestion économique des produits, principalement le marketing.

- **Domain Requirements engineering**
  Détermine les exigences communes et variables de la ligne de produit (ou famille de produit)

- **Domain design**
  Détermine l'architecture de référence de la ligne de produits. Celle-ci fournit une structure commune, de haut niveau, pour tous les produits de la ligne de produits.

- **Domain realisation**
  Conception détaillée et implémentation des composants réutilisables.

- **Domain Testing**
  Tests et validation des composants réutilisables.

Ces différents sous-processus génèrent des domain artefacts, ce sont des exigences, des définitions d'architecture et des composants réutilisables à transmettre à l'application engineering.

Comme dans bien d'autres développements, ces étapes d'ingénierie d'exigence, d'architecture, de réalisation et de tests peuvent se suivre lors de plusieurs cycles.

1.2.2 Application Engineering

Ce processus représente l'analyse et le développement de produits (applications) à partir de la plateforme.

Chaque application intégrera les fonctionnalités dont elle a besoin parmi celles offertes par la plateforme. Elle exploitera la variabilité de cette plateforme (l'espace de liberté que les composants laissent aux applications) afin de répondre à ses besoins spécifiques.

Ce processus se compose de 4 sous-processus, ceux-ci vont utiliser les domains artefacts fournis par le domain engineering :

- **Application Requirements Engineering**
  Détermine les exigences de l'application. Les écarts entre les exigences de la ligne de produits et du produit devront être détectés lors de ce processus. Les exigences pourraient alors être adaptées afin de réduire les écarts et donc améliorer la réutilisabilité de la plateforme.

- **Application Design**
  Détermine l'architecture de l'application. L'architecture de référence du domaine est utilisée pour instancier l'architecture de l'application. Cela consiste à sélectionner quels éléments architecturaux de la ligne de produits doivent être intégrés à l'application, ainsi que d'y incorporer des adaptations spécifiques au produit.

- **Application Realisation**
  Réalisation de l'application. Cela consiste à sélectionner parmi les composants réutilisables du domaine ceux à utiliser dans l'application, ainsi que réaliser les composants propres à l'application et intégrer les composants propres et réutilisables dans l'application.

- **Application Testing**
  Tests de l'application

1.2.3 Réutilisabilité « pro-active »

La réutilisabilité n'est pas exclusivement liée aux SPLs. De nombreuses applications conçues hors du cadre des SPLs l'utilisent également, mais souvent de manière opportuniste (Ex : Un concepteur découvre qu'un composant d'une application (routine, objet OO,…) peut lui rendre service, alors il décide de l'intégrer). Par contre dans le cadre des SPLs, les composants sont volontairement conçus pour être réutilisés.

1.3 Avantages des SPLs.

Voici différents avantages des SPLs, expliqués en détails dans le livre de Pohl et al. [1] .

1.3.1 Augmentation de la qualité

L'utilisation fréquente des outils et composants communs lors de la fabrication de nombreux produits permet de les évaluer constamment et de découvrir plus facilement leurs défauts. A la longue, ces outils et composants deviennent donc de plus en plus fiables et performants.

1.3.2 Maitrise des estimations de coûts et délais

Lors de chaque étape du développement d'un produit, l'informaticien est confronté à bon nombre de risques. L'estimation des délais et du coût est donc peu fiable car soumise à un risque élevé.
Par contre, si le produit résulte de l'assemblage de composants déjà développés, selon des procédures bien définies, le risque est nettement inférieur. L'estimation est donc plus fiable.

1.3.3 Réduction des coûts de développements.

L'élaboration de la plateforme a un coût non négligeable. Pour un nombre limité de produits, le coût de production de ces produits est plus important que la production de ces produits de façon isolée puisqu'il intègre ce coût d'investissement.

Dans le schéma ci-dessous, la droite représentant le coût de production de produits dans le cadre d'une SPL est en pointillée. La droite en trait plein représente le coût de production de produits de façon isolée.

Ce schéma illustre bien que jusqu'à un certain nombre de produits, le coût accumulé de production dans une SPL est supérieur à celui hors SPL.

En effet, le coût accumulé de production représente le coût total de production de tous les produits fabriqués depuis le début de la production.

Hors SPL, la production de chaque produit à un cout similaire, ce qui implique que le coût cumulé augmente proportionnellement au nombre de produits.

Par contre dans une SPL, le coût cumulé de production des premiers produits est élevé puisque l'on y intègre le coût de développement des composants réutilisables (« Up-Front Investment ») et le cout de production des premiers produits. Mais une fois les composants développés et validés, l'ajout de nouveau produit se fera à un coût réduit, puisqu'il s'agira en grande partie de l'intégration de composants déjà développés.

Le coût cumulé de production dans une SPL augmente aussi en suivant l'augmentation du nombre de produits, mais dans une moindre mesure.

Donc, si la production dépasse un certain nombre de produits, le coût cumulé de développement des produits est inférieur dans le cadre d'un SPL à celui hors SPL, tout comme le coût moyen d'un produit.

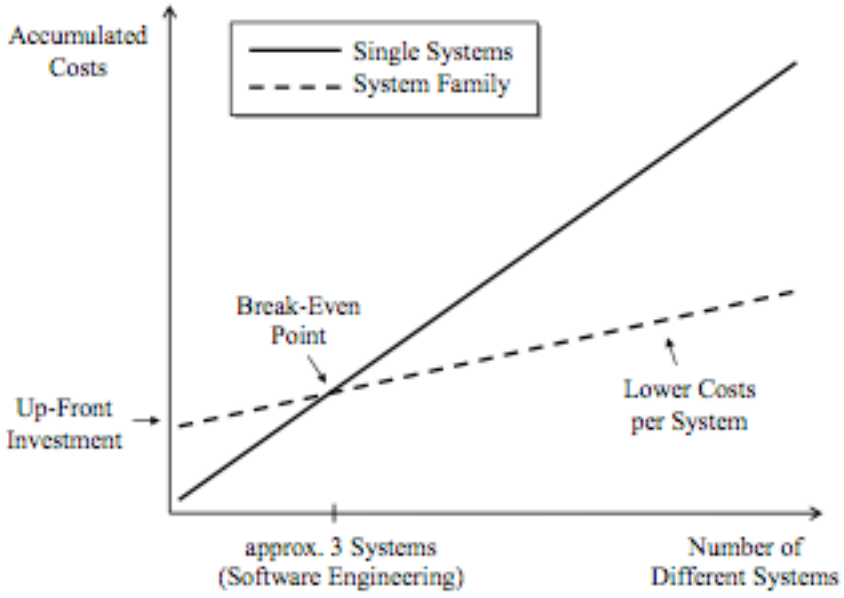


FIGURE 1.2 **Coût de développement de N systèmes hors SPL comparé au coût de développement de N systèmes d'une SPL (Pohl et al. [1], page 10).**

1.3.4 Réduction du délai de commercialisation

Par rapport à une production indépendante de produit, la mise en commun de composants permet de gagner du temps. En effet, l'élaboration de la structure et des composants a un coût temporel supérieur, mais ce temps « perdu » au départ permettra de réduire le temps de développement des produits.

Comme l'indique la figure 1.3, le développement de la plateforme prend un temps non négligeable. Le temps nécessaire à la commercialisation des premiers produits est donc plus important que le temps nécessaire à la commercialisation de produits isolés.

En effet, dans le cadre d'un SPL, le temps de développement de la plateforme s'ajoute au temps de développement des premiers produits.

Par contre, comme indique le schéma, le temps de développement de produit hors d'une SPL est similaire quel que soit le nombre de produits (droite pleine) alors que le temps de développement des produits dans une SPL (droite pointillée) est décroissant selon le nombre de produits.

En effet, plus le nombre de produits est élevé, plus le temps nécessaire à la production des composants va être amorti sur un grand nombre de produits. Le temps moyen de production d'un produit est donc moins élevé.

Cette décroissance du temps de développement dans une SPL (droite pointillée) est due à la réutilisabilité des composants. Plus la production de produits est grande, plus le nombre de composants développés et pouvant être réutilisés augmente.

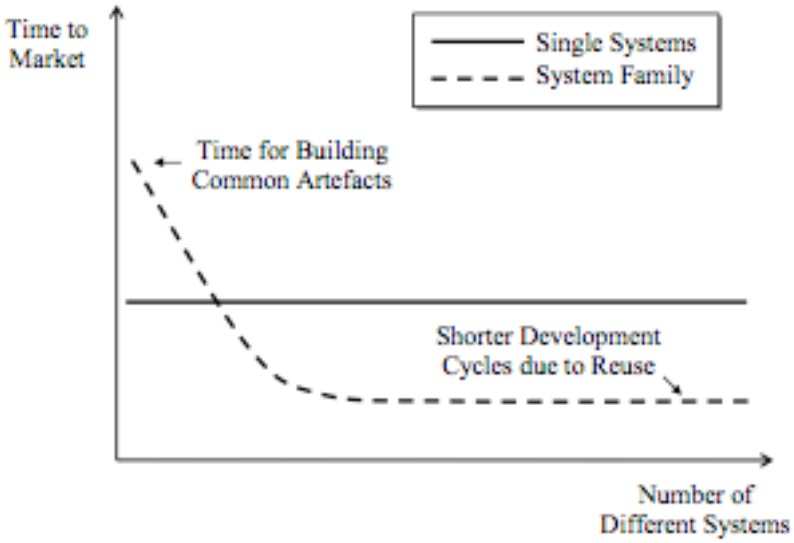


FIGURE 1.3 **Comparaison du délai de commercialisation de produits membres d'une SPL et avec des produits identiques développés indépendamment (Pohl et al. [1], page 11).**

1.3.5 Réduction des coûts de maintenance/évolution.

Quand un composant de la plateforme est modifié, ce changement peut être propagé à tous les produits utilisant ce composant, sans connaître obligatoirement les détails de chaque produit.

L'ajout d'un nouveau composant le rend aussi facilement disponible à tous les produits dérivés.

1.3.6 Gestion de la complexité

L'époque où les utilisateurs souhaitaient de simples programmes recevant quelques données en entrées et fournissant un résultat est révolue. Le nombre de fonctionnalités d'un logiciel, la complexité de la logique sous-jacente et l'interconnexion entre différentes applications et différents composants hardware génèrent une complexité quasiment ingérable pour un système développé de A à Z isolément.

Heureusement, les SPLs diminuent considérablement la complexité, chaque problème pouvant être délégué à un sous-système déjà développé.

C'est le cas, par exemple, d'une application bancaire nécessitant, entre autres, la connaissance métier, la gestion des transactions et la gestion de la sécurité. Très souvent, les informaticiens vont devoir développer la logique métier propre à leur application, mais vont pouvoir intégrer des modules de gestion de la sécurité et un gestionnaire de transactions, ce qui diminue fortement la complexité du développement, les délais, les coûts, les risques et les connaissances nécessaires.

1.4 Difficultés

Malgré leurs nombreux avantages, les SPLs ont tardé à se développer dans le monde informatique.

Quelles sont les raisons d'échecs ou de réticences envers les SPLs ? Le facteur principal est le coût, le délai et les ressources nécessaires au domain engineering.

Manque de financement et de temps

Dans une société ou chaque entreprise veut être présente la première et où obtenir un crédit s'avère difficile, se lancer dans une phase de domain enginnering est délicat. Les décideurs, pressés par le marché, ne disposent que trop rarement du temps et du financement nécessaires à la phase du domain engineering : le marché impose une commercialisation des premiers produits le plus tôt possible et à moindre coût. Or, la section précédente (1.3.1 et 1.3.4) illustre bien que les coûts financiers et temporels des premiers produits sont plus élevés. Les décideurs doivent donc disposer du temps et du financement nécessaires pour réaliser l'investissement.

Risques menaçant la rentabilité

Une fois la décision d'investissement validée, rien n'est gagné.

Une connaissance approfondie du domaine s'avère primordiale afin de produire en masse un produit répondant aux attentes.

Plusieurs risques pèsent sur le résultat :

- **Connaissances insuffisantes du domaine** : Le délai et le coût de domain engineering risquent alors d'être dépassés et/ou le résultat risque d'être peu convainquant suite à un temps d'apprentissage élevé et de mauvaises spécifications des composants de base.
- **Instabilité du domaine** : Inutile d'investir dans une phase de domain engineering si les contraintes du domaine changent peu de temps après son analyse.
- **Demande trop diversifiée :** L'intérêt des SPLs étant la réutilisabilité de composants, la production de produits avec peu de points communs ne permettraient pas cette réutilisabilité.
- **Demande trop réduite :** Comme illustré sur les figures 1.2 et 1.3, plus la production est élevée, plus les SPLs sont rentables. Une demande réduite nuira donc à la rentabilité.

1.5 Conclusion

Ce chapitre introduit les concepts liés aux SPLs et la variabilité. Mais comment représenter cette variabilité des SPLs ? Le chapitre suivant présente des notations graphiques permettant de répondre à cette question.

2. Feature Diagrams.

Le chapitre précédent décrivait les SPLs et le concept de variabilité des SPLs. Ce chapitre va présenter les feature diagrams (parfois appelé feature models), ce sont des langages graphiques permettant de représenter la variabilité des SPLs.

Ce chapitre présente tout d'abord quelques définitions du terme « feature », le terme clé dans la représentation de la variabilité ainsi que les origines des feature fiagrams.

Ensuite, les concepts et la syntaxe des feature diagrams sont présentés, tels que les nœuds, les liens hiérarchiques, les attributs et les contraintes.

La notion de clonage sera ensuite décrite, distinguant le feature model de son implémentation.

Enfin, nous aborderons les concepts des feature diagrams nécessitant la distinction entre modèle et produit.

La suite de ce chapitre décrit les différents concepts des feature diagrams et est inspirée de la présentation de feature diagrams de A. Hubaux [23] et le présentation de Kang [3].

2.1 Feature et Feature Diagrams

Le terme clé est celui de Feature, dont voici quelques définitions de divers auteurs, synthétisées dans Classen et al. [4].

Selon Kang et al [5], un « **Feature** » est: « A prominent or distinctive user-visible aspect, quality, or characteristic of a Software system or systems. »

Selon Bosch et al [6]: « A logical unit of behaviour specified by a set of functionnal and non-functionnal requirements.»

Ou encore selon Chen et al. [7]: « A product characteristic from users or customer views, which essentially consists of a cohesive set of individual requirements.»

Un feature est donc une abstraction, pour l'utilisateur et le développeur, permettant de modéliser la variabilité.

Afin de mieux comprendre ce terme, voici un exemple simple, imaginons une gamme « d'ordinateur de bord » pour VTT. Tous ces produits disposent d'un compteur de vitesse, mais certains disposent d'un capteur de fréquence cardiaque et d'autres d'un GPS, voire des deux. Le compteur de vitesse, le capteur de fréquence et le GPS peuvent être considérés comme des « features ».

Le feature diagram est un graphe, représentant les types de produits possibles de la ligne de produits. Dans l'exemple de l'ordinateur de bord, le graphe indiquerait que le feature compteur doit être présent et que les features capteur de fréquence et GPS peuvent être ajoutés.

Bien entendu, comme l'indique les différentes définitions ci-dessus, un feature ne se limite pas à une fonctionnalité, il peut aussi s'agir d'un aspect visible, d'un niveau de qualité, d'un composant logiciel, d'un ensemble d'exigences.

2.2 Un peu d'histoire

La première définition publiée des feature diagrams date de 1990, dans l'article Feature-Oriented Domain Analysis de Kang et al. [5]

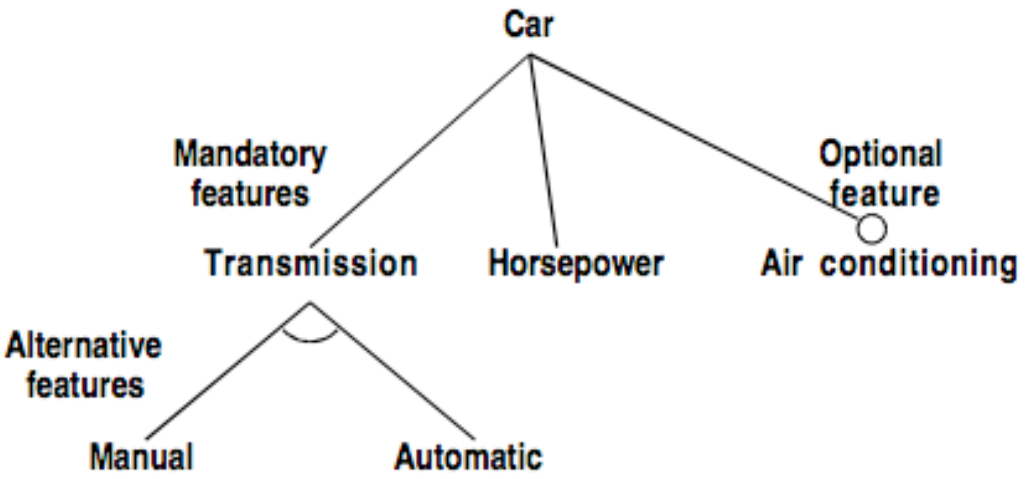


Figure 2.1 Feature Diagram selon FODA [5]

Cette première définition décrivait les concepts de base des feature diagrams, tels que les features et la hiérarchie de features (présentés au point 2.3). Mais ces concepts furent ensuite étendus. Actuellement, aucun standard décrivant la modélisation des feature diagrams ne s'est imposé.

La figure suivante schématise sous forme d'arbre différentes versions de feature diagrams. La version racine est FODA, elle a ensuite été étendue par diverses versions, dont certaines ont également été étendues par la suite.

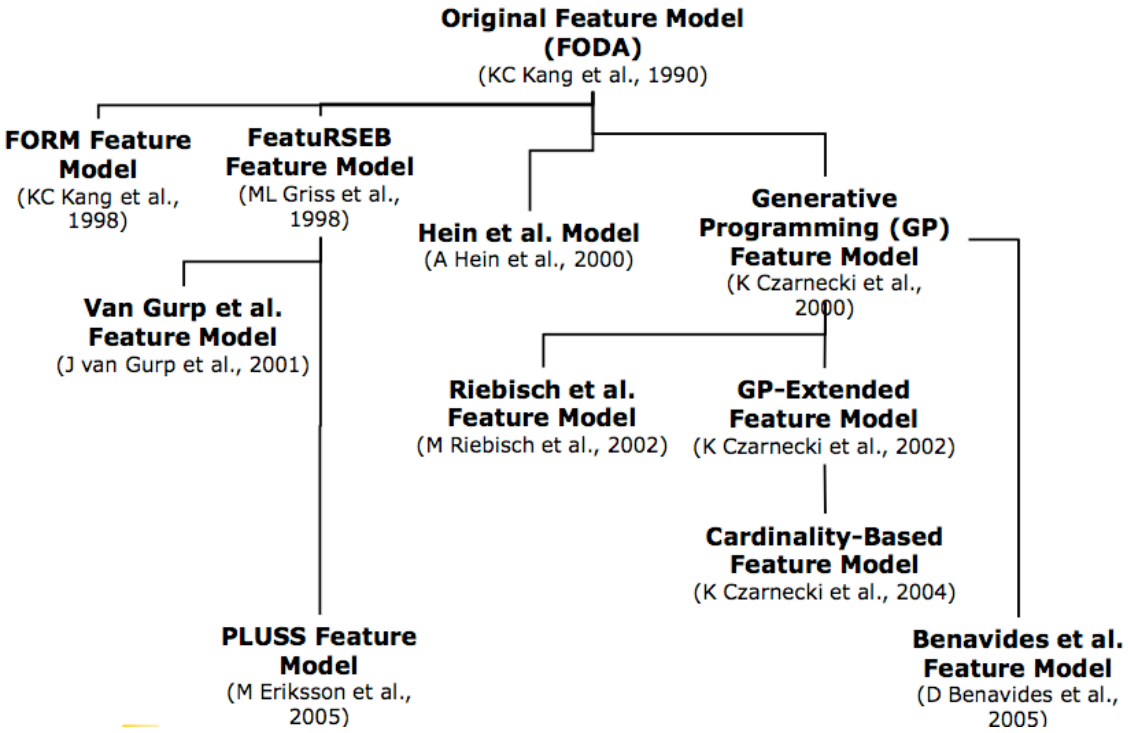


Figure 2.2 Généalogie des feature diagrams selon Kang [3]

2.3 Composants de base d'un Feature Diagram.

2.3.1 Les nœuds (ou « node » ou « feature »).

Les nœuds représentent les features, généralement par un rectangle contenant un libellé. Ce libellé doit être unique dans le Feature Diagram, il correspond au nom du feature. (Czarnecki et al., [9,10,11], Riebisch et al. [12,13])

Certains langages diffèrent très légèrement, ils n'utilisent pas les contours du rectangle. (Kang et al. [5], Griss et al. [26])

2.3.2 Les liens.

Chaque lien représente une relation binaire entre deux features (feature pris au sens de concept réel qu'il représente).

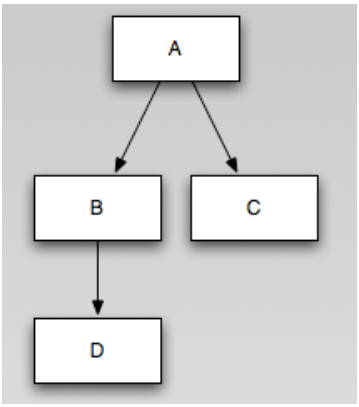
Cette relation (ou décomposition) se lit généralement de haut en bas. Le feature supérieur est le parent (ou « superfeature »), l'autre est l'enfant (« childfeature » ou « subfeature »).

La relation de parent-enfant peut être généralisée, on distingue alors :

- Parent (direct superfeature)
- Ancêtre (indirect superfeature, ancestor)
- Enfant (direct subfeature)
- Descendant (indirect subfeature)

Exemple, dans le schéma ci-dessous :

- A est parent de B et C, A est aussi la racine
- B et C sont enfants de A
- D est descendant de A
- A est ancêtre de D



2.3.3 Le graphe.

Le graphe formé par les nœuds et liens vérifie plusieurs propriétés :

- Orienté (« directed »): source et cible des liens sont définis (généralement par la lecture de haut en bas)
- Acyclique (« acyclic »): aucun chemin dans le graphe ne contient de cycle
- Contient une et une seule racine (un nœud qui n'est la cible d'aucun lien)

Ce graphe est donc un DAG (directed acyclic graph), contenant une seule racine.

Habituellement, c'est aussi un arbre (« tree ») : un DAG dans lequel chaque nœud excepté la racine a exactement 1 parent.

2.4 Décomposition

Comme vu dans la section précédente, un parent peut se décomposer en plusieurs enfants.

Chacun de ces enfants est soit :

- **Optionnel** (représenté avec un cercle vide au sommet)
- **Obligatoire** (par défaut, parfois représenté avec un cercle plein au sommet)

Plusieurs types de décompositions existent, basés sur la logique booléenne ou sur les cardinalités

- **And**
- **Or**
- **Xor**
- **[x..y]**

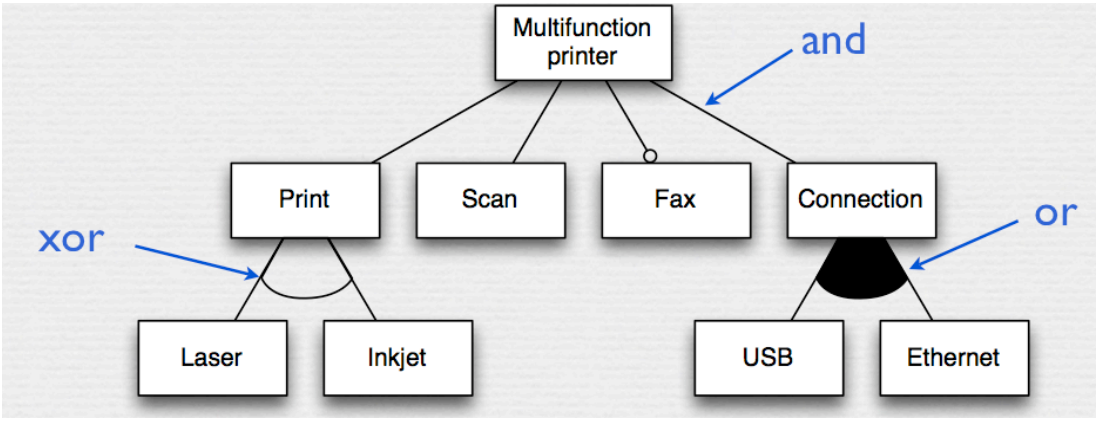


Figure 2.3 Feature diagram avec 3 types de décompositions (and, or, xor) (A Hubaux [23])

La figure 2.3 présente des exemples de décomposition «and», « or » et « xor » afin d'illustrer leur syntaxe généralement utilisée dans les feature diagrams.

Les points suivants décrivent une signification généralement admise des différentes décompositions, notamment dans le cas d'une présence de features optionnels dans la décomposition. Une sémantique exacte est présentée dans la description de TVL. Evidemment, d'autres normes définissent des sémantiques différentes.

Tous les enfants d'un feature présents dans le produit, dont la décomposition est de type AND doivent se trouver dans le produit.

2.4.1 Décomposition AND

Attention, le caractère optionnel d'un feature a une influence dans ce cas. Il convient de définir une sémantique indiquant les priorités des décompositions et des caractères optionnels.

Dans l'exemple de la figure 2.3, « Print », « Scan », « Fax » et « Connection » se trouvent sous une décomposition AND, mais « Fax » est optionnel. Si la sémantique donne priorité au caractère optionnel, un produit contenant « Print », « Scan » et « Connection » serait valide, que « Fax » soit présent ou pas. Par contre, l'absence d'un des composants obligatoires ne serait pas valide.

2.4.2 Décomposition OR

Si les enfants d'un feature, présent dans le produit, ont une décomposition de type OR, alors au moins 1 enfant doit être présent dans le produit.

Exemple, dans la figure 2.3, si le produit contient « Connection », alors il doit aussi contenir « USB » ou « Ethernet » (ou les deux).

Dans ce cas, l'utilisation de feature optionnel est généralement déconseillée.

2.4.3 Décomposition XOR

Ce cas est similaire au OR, mais il s'agit d'un « ou exclusif ». Dans le cas de la figure 2.3, le feature « Print » peut être décomposé en « Laser » ou « Inkjet », mais pas les deux.

2.4.4 Décomposition basée sur les cardinalités

La décomposition spécifie des cardinalités, d'une borne inférieure et d'une borne supérieure, sous la forme : [min..max]

Une décomposition de [x..y] signifie qu'au minimum x et au maximum y enfants doivent être présents dans le produit.

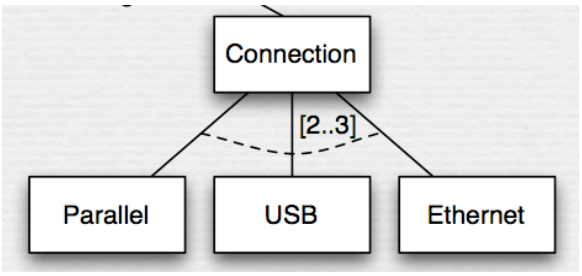


Figure 2.4 Feature diagram avec cardinalités (A Hubaux [23])

Dans la figure 2.4, si « Connection » est présent dans un produit, la décomposition de ce feature doit comporter entre 2 et 3 subfeatures, parmi {« Parallel » , « USB », « Ethernet » }.

Cette notation offre donc une expressivité supérieure à celle la forme booléenne. Bien entendu, elle fournit une représentation équivalente à chaque décomposition booléenne :

AND devient [N..N], le produit doit contenir entre N et N éléments parmi N subfeatures, donc tous les subfeatures.
OR devient [1..N], ce qui est vérifié si au moins un des subfeatures est présent.
XOR devient [1..1], un et un seul subfeature doit être sélectionné.

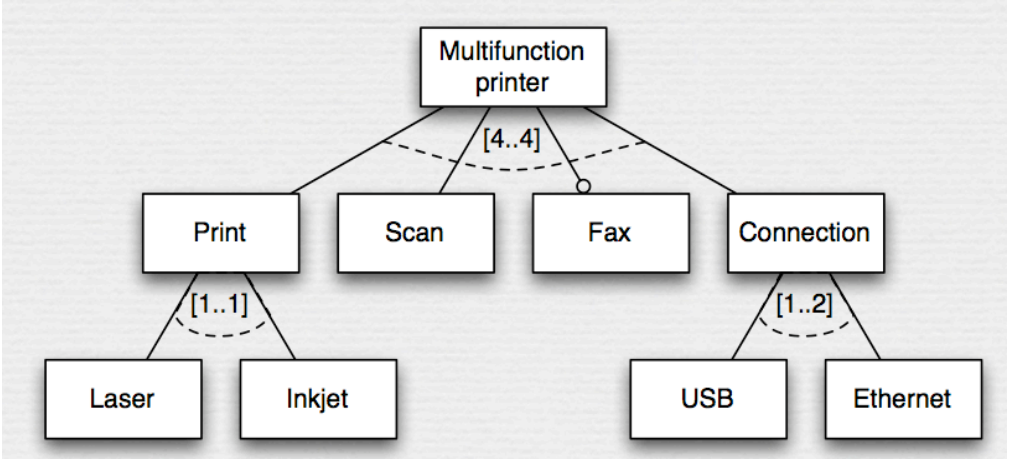


Figure 2.5 **Feature diagram avec cardinalités exprimant des décompositions booléennes (A. Hubaux [23])**

Le schéma de la figure 2.5 est équivalent à celui de la figure 2.3. Les décompositions booléennes ont été traduites sous forme de cardinalités.

Les features optionnels peuvent modifier les contraintes exprimées par les cardinalités. Une discussion sur ce sujet sera présentée lors de la présentation des extensions de TVL.

Des auteurs dont Czarnecki et al. [11], Riebisch et al. [12] décrivent les décompositions exprimées sous forme de cardinalités.

2.5 Contraintes

Deux formes simples de contraintes textuelles sont généralement exprimées directement dans le Feature Diagram :

- A « **Requires** » B : signifie que si le feature A est sélectionné, le feature B doit également être sélectionné, cela équivaut à la proposition : A => B
- A « **Excludes** » B : signifie que les Features A et B sont mutuellement exclusifs, si l'un est sélectionné, l'autre ne peut pas l'être, cela équivaut à ¬ (A ∧ B)

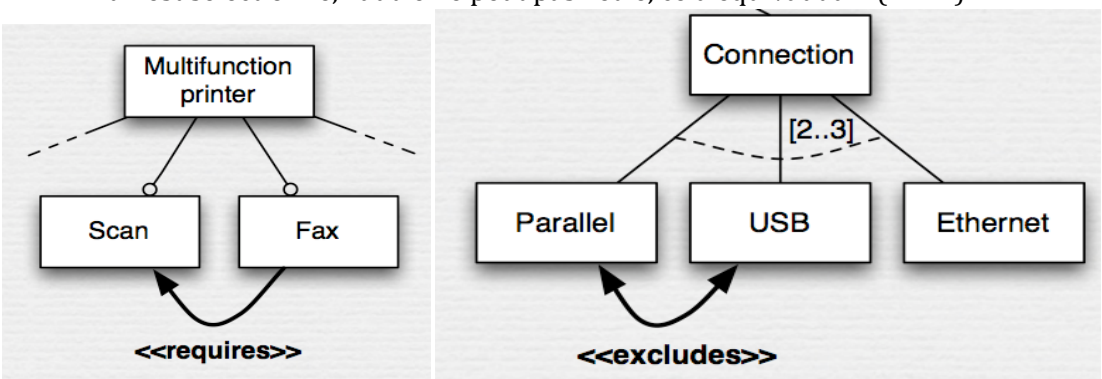


Figure 2.6 **Extraits de Features diagrams avec contraintes requires et excludes ((A. Hubaux [23])**

Dans la figure 2.6,

- la présence du feature « Fax » implique la présence du feature « Scan ».
- le feature « Connection » ne peut pas disposer à la fois d'un feature « Parrallel » et d'un feature « USB ».

La logique des propositions permet également d'exprimer des contraintes, celles-ci sont généralement décrites dans le feature diagram.

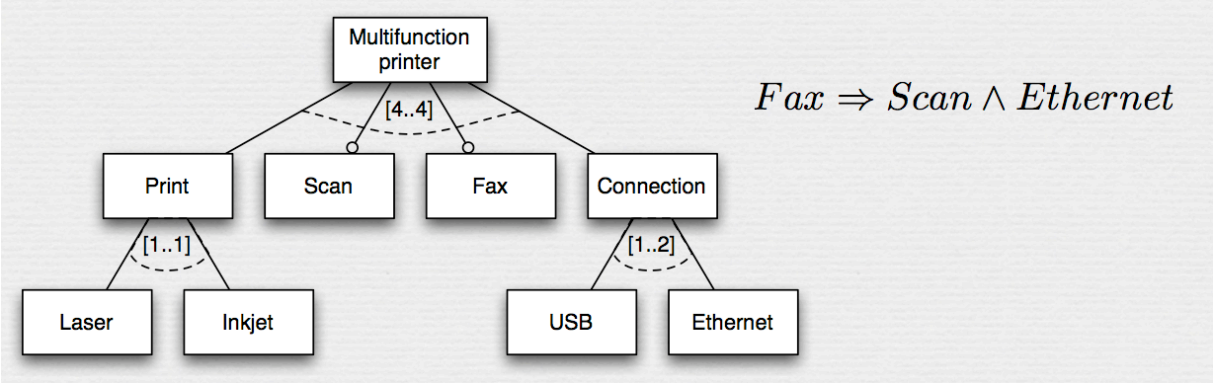


Figure 2.7 Feature diagram avec contrainte (logique des propositions) (A.Hubaux [23])

La contrainte exprimée dans figure 2.7 signifie que si le feature « Fax » est présent, les features « Scan » et « Ethernet » doivent l'être également.

La logique des propositions permet d'exprimer des contraintes beaucoup plus complexes que les clauses « requires » et « excludes » ne le permettent, et ce de manière relativement claire, sans surcharger le Feature Diagram.

Différents auteurs décrivent l'utilisation de contraintes, notamment ML Griss et al., [26]; Czarnecki et al. [9].

2.6 Attributs

Les attributs sont généralement de deux types :

- Primitifs (int, real, bool, string,…)
- Enumération

Ces attributs permettent d'exprimer des caractéristiques des features, qui pourront être définis par les produits. Cela offre donc une variabilité supplémentaire.

Les attributs de type énumération vont plus loin. Ils permettent de réduire la taille du feature diagram en exprimant certaines caractéristiques des features sous forme d'attributs plutôt que sous forme de décomposition. Cette astuce n'est toutefois possible que pour les features terminaux (feature sans enfants).

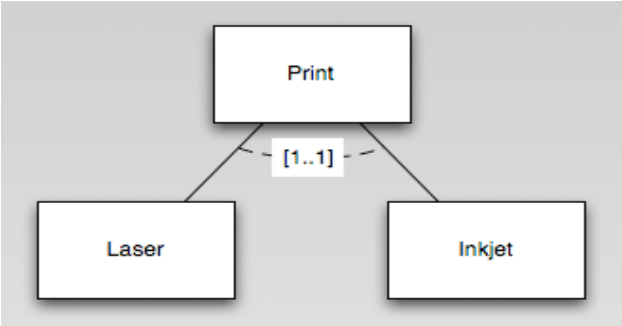


Figure 2.8 Feature Diagram sans énumérations, utilisation de sous-feature à la place d'une énumération.

Dans l'exemple de la figure 2.8, l'ajout d'un attribut « Type » de type énumération (Laser, InkJet) dans le feature « Print » permettrait de supprimer les features « Laser » et « Inkjet ». Le schéma deviendrait comme décrit en figure 2.9.

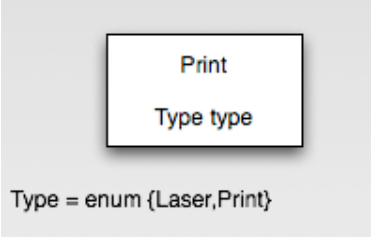


Figure 2.9 Feature diagram avec énumération, utilisation de celle-ci à la place de feature terminaux

Plusieurs auteurs, dont Benavides [8], proposent une syntaxe de définition des attributs.

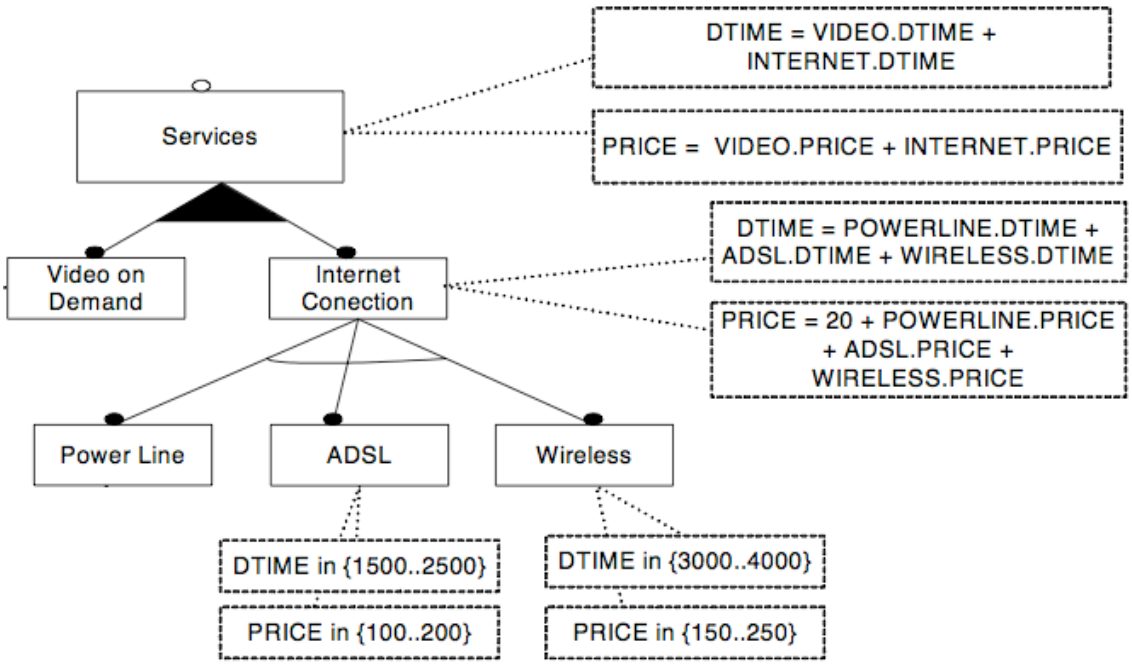


Figure 2.10 Exemple de Feature diagram avec attributs selon Benavides

Les attributs dans la figure 2.10 sont indiqués à l'intérieur de cadres associés au feature auquel ils se rapportent, par exemple : DTIME et PRICE pour ADSL (Nous ne détaillons pas les expressions de définition de contraintes sur les attributs).

2.7 Feature Model et instance

Le feature model est un modèle décrivant l'ensemble des produits valides appartenant à une ligne de produits.

Cette notion de feature model est assez proche de celle de feature diagram, le feature diagram étant un diagramme permettant de représenter un feature model.

Par analogie aux langages orientés objet comme Java ou C#, le feature Model peut être vu comme la classe, alors que le produit peut être vu comme une instance de cette classe.

Ce principe s'applique également au niveau des features. Un feature du feature model est instancié dans le produit. Le feature définit les types d'attributs, les contraintes et les cardinalités de décomposition du feature, alors que l'instance de ce feature dans un produit va pouvoir spécifier des valeurs d'attributs qui lui sont propres et sélectionner quels sous-features seront instanciés dans sa décomposition, tout en respectant les contraintes imposées par le feature du feature model.

Certains feature diagrams permettent de représenter un feature model dans lequel certains features sont instanciés plusieurs fois, on parle alors de clonage (terme utilisé dans le vocabulaire TVL). Ces feature diagrams utilisent alors des cardinalités de feature.

Le clonage et les cardinalités de feature sont présentés au point suivant, une description plus complète associée à TVL est décrite au point 5.1 et détaillée dans l'article de Michel et al. [22]

2.8 Cardinalités de Feature et clonage

Les cardinalités de feature ont généralement une notation similaire aux cardinalités de groupe : [min..max], mais celle-ci sont placées à proximité du feature et non d'une décomposition.

Cela signifie que le feature peut être instancié dans le produit au moins « min » fois et au maximum « max » fois, tout en maintenant les autres règles du modèle.

La cardinalité « max » peut être non-bornée, par exemple [1..*] signifie que le feature peut être instancié un nombre infini de fois.

Un « min » = 0 signifie que le feature est optionnel, quelle que soit la valeur de « max ».

Le comportement des cardinalités de feature n'est pas trivial, ces cardinalités influencent les autres concepts :

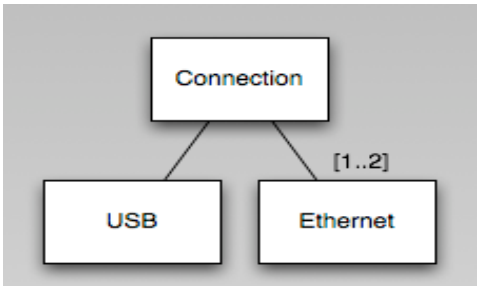


Figure 2.11 Extrait d'un feature diagram avec cardinalités de feature

Dans l'exemple de la figure 2.11, un produit dans lequel « Connection » est sélectionné doit contenir un « USB » et de 1 à 2 « Ethernet ». Ce cas est simple car il s'agit d'une décomposition « and », mais que faire s'il s'agissait d'une décomposition « or », « Ethernet » peut-il être ignoré ou doit-il être présent ?

Tout dépend de la priorité que l'on donne aux différents opérateurs (décomposition et cardinalités de feature). Une discussion plus axée sur les interactions entre les cardinalités de feature et les principaux concepts de TVL sera présentée dans le chapitre 4.

Examinons le cas du clonage :

Toutes les instances d'un feature (appelées aussi clones du feature) doivent respecter les mêmes règles héritées du modèle, tels que le type et le domaine des attributs, les contraintes et les cardinalités de décomposition.

Toutefois, en général, il s'agit bien d'instances différentes. Ces instances ont alors leur propre état et peuvent sélectionner différents sous-features dans leur décomposition.

Voici un exemple de modèle disposant de cardinalités de feature, le feature « Room » possède des attributs et des sous-features :

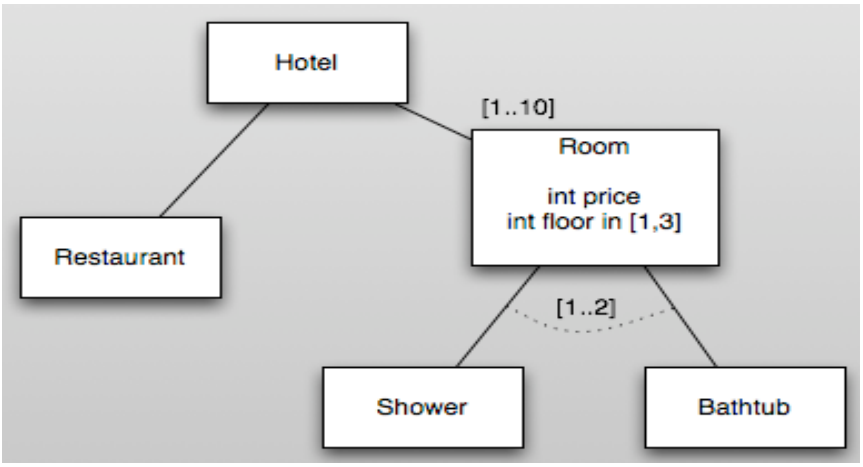


Figure 2.12 Feature Clonable avec attributs et sous-features.

Voici le schéma illustrant un produit instanciant ce modèle :

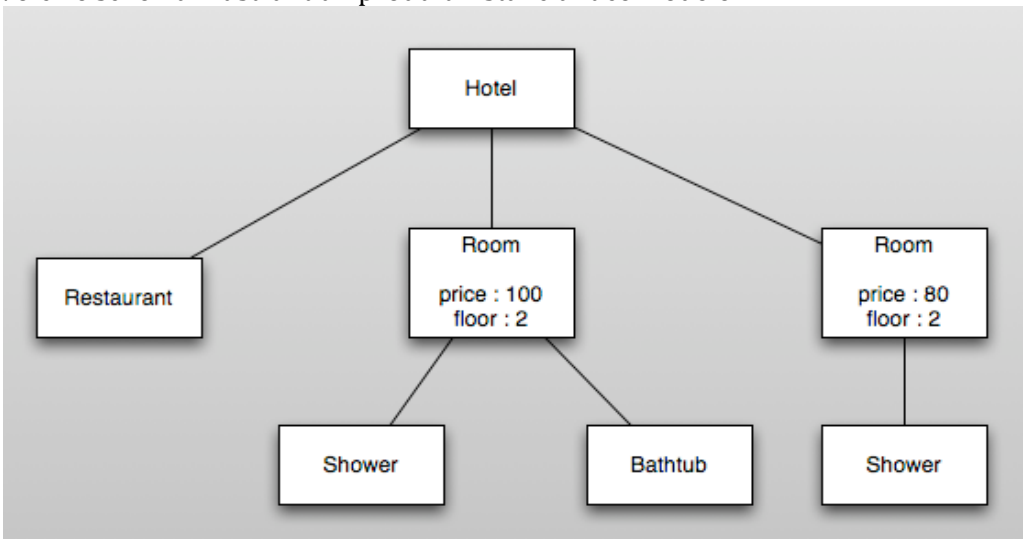


Figure 2.13 Produit avec 2 instances d'un même feature, avec état différents.

Dans ce schéma, les « Room » sont des clones du feature « Room ». Ce schéma est valide, en effet :

- la cardinalité de feature [1..2] est respectée : 2 clones sont présents
- pour chaque clone, les types et valeurs des attributs sont conformes aux modèles, **bien que les valeurs soient différentes**
- pour chaque clone, la décomposition et la cardinalité de groupe sont respectées, **bien que différentes dans chaque clone.**

Plusieurs auteurs se sont intéressés aux cardinalités de feature, tels que : Czarnecki et al. [10,11], Riebish et al. [12], Michel et al. [22]

2.9 Inclusions

Le concept d'inclusion (ou référence) permet d'éviter la duplication de features dans le diagramme. Un petit exemple permet de comprendre ce concept (figure 2.8).

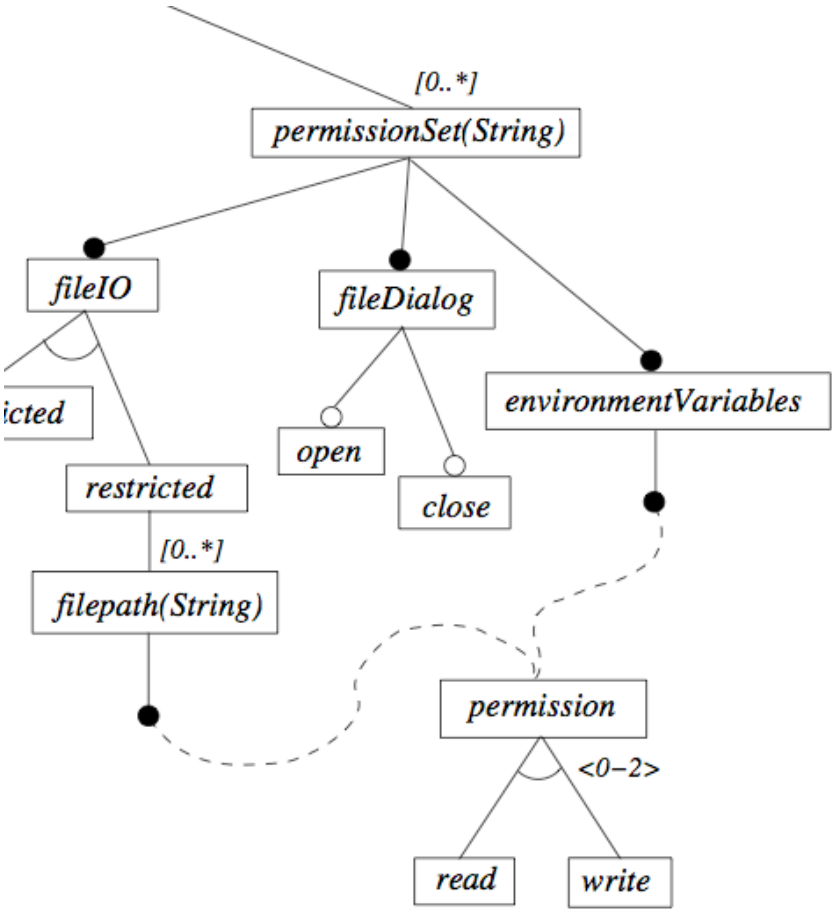


Figure 2.8 **Feature Diagram avec références (Czarnecki et al., 2004) [9]**

Dans ce feature diagram (fig. 2.8), FilePath et environmentVariables se composent tous les deux d'un feature Permission.

- Il s'agit bien du même feature
- La variabilité en dessous de « Permission » est identique dans les deux cas
- **ATTENTION,** lors de l'instanciation d'un produit, les choix effectués concernant la décomposition de « Permission » sous « FilePath » sont indépendants de ceux effectués pour la décomposition de « Permission » sous « environnementVariables ». (Ce qui signifie dans le monde réel, que les permissions accordées à un utilisateur sur un fichier sont différentes de celles accordées sur les variables d'environnement).

Ce mécanisme d'inclusion permet donc de simplifier et de réduire la taille des feature diagrams en permettant une factorisation.

2.10 Conclusion

Ce chapitre a introduit les feature diagrams ainsi que leurs concepts. Toutefois, vu les différentes versions de feature diagrams, cette présentation est générale, elle n'a pas pu donner une sémantique exacte. TVL sera présenté dans le chapitre 4, la syntaxe et les différents choix au niveau de la sémantique y seront alors expliqués, notamment concernant les interactions entre les features optionnels et les décompositions, ainsi que la sémantique des cardinalités de features.

3. Analyse de langages

Ce chapitre vise à rappeler brièvement ce que représente un compilateur (interpréteur), son rôle, ainsi que les étapes clés de la compilation des langages. Il est inspiré principalement du livre « Compilateurs » de Grune et al. [15] ; du tutoriel Théorie des langages de l'Université de Bretagne [16] et du cours de Syntaxe & Sémantique du Professeur P-Y. Schobbens [17].

TVL étant un langage de variabilité au format textuel, il est concerné par les méthodes classiques d'analyses de langage.

3.1 Aperçu global

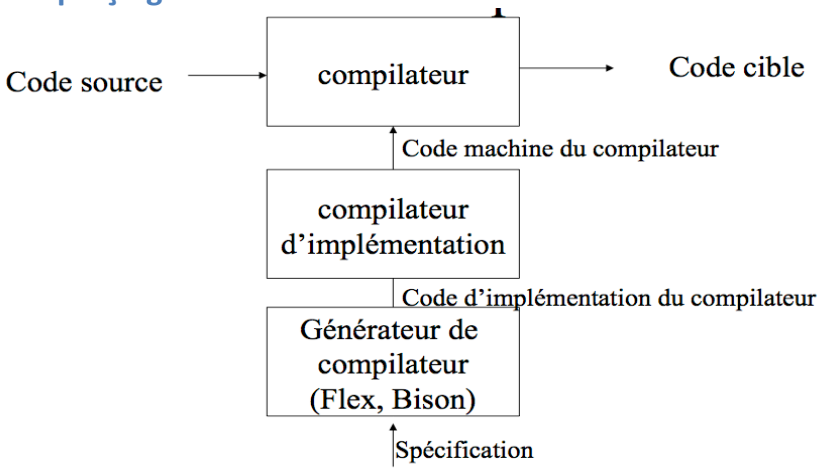


Figure 3.1 Illustration du rôle d'un compilateur (cours syntaxe et sémantique du prof. P-Y Schobbens [17])

Comme représenté dans la figure 3.1, un compilateur est généralement un outil permettant de traduire un code source, exprimé dans un langage source, dans un code cible, exprimé dans un langage cible. La lecture du code source et sa représentation dans une structure interne au compilateur sont appelées « partie **avant** » du compilateur, tandis que la traduction de cette représentation interne dans un langage cible est appelée « partie **arrière** ».

Différentes variantes existent pour la partie « arrière » : génération de code cible, interprétation,… Dans le cadre de TVL, ce mémoire s'intéressera principalement à la partie « avant », dont voici une décomposition.

Comme l'indique la partie inférieure de la figure 3.1, un compilateur est habituellement généré à l'aide d'un générateur de compilateur. Il s'agit d'un outil permettant de générer le code source du compilateur à partir d'une spécification (Définition des **unités lexicales** et **grammaire**, voir 3.2 et 3.3). Ce code source est ensuite compilé par un autre compilateur, appelé compilateur d'implémentation, afin d'obtenir l'exécutable du compilateur que l'on souhaite créer.

La figure 3.2 schématise les grandes étapes de la compilation de langages. La partie « avant » reçoit le code source et effectue une analyse lexicale, puis une analyse syntaxique et enfin une analyse sémantique. Ces analyses permettent d'exprimer le code source dans une représentation interne qui pourra ensuite être utilisée dans la partie « arrière » du compilateur.

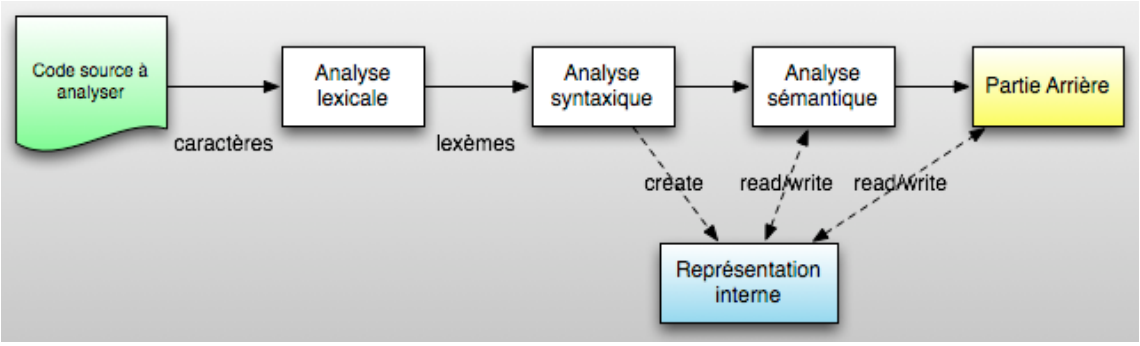


Figure 3.2 Structure « avant » d'un compilateur

Les différents concepts présentés sur ce schéma sont détaillés dans les points suivants.

3.2 Analyse lexicale

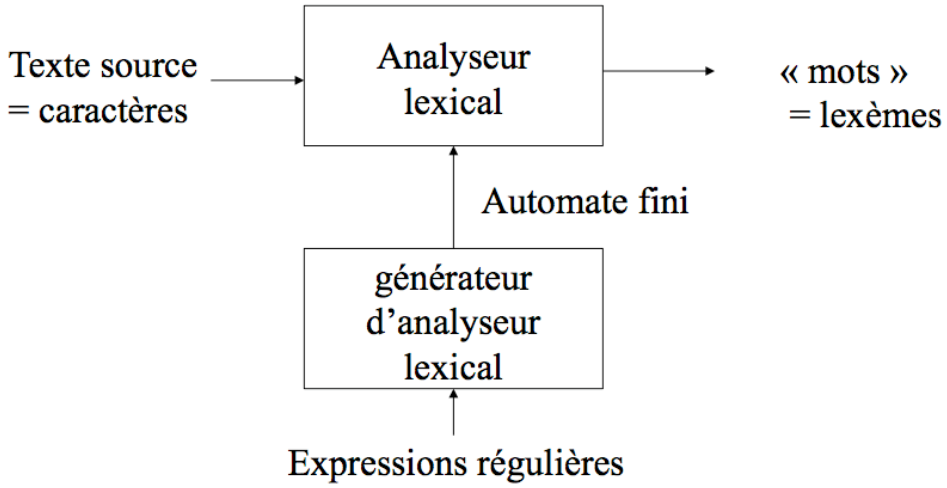


Figure 3.3 Analyse lexicale

L'analyse lexicale consiste à analyser le texte source, lu sous forme de caractères, afin de générer une suite de « mots » (lexèmes).

Dans un langage informatique, les lexèmes sont par exemple les chaines de caractères, les entiers, les variables,…

Généralement, des expressions régulières permettent de regrouper les caractères du flot d'entrée et de déterminer quel lexème ils représentent.

Par exemple, si les expressions régulières définissent les trois unités lexicales suivantes :

- IDENT : [a-z]*
- INT : [1-9][0-9]*
- AFFECT : =

Si le flot d'entrée est « nbr = 10 », les expressions régulières permettent d'identifier la suite de lexèmes suivante : IDENT AFFECT INT.

Cette transformation est effectuée par un automate fini généré par le générateur d'analyseur lexical, tel que lex, flex, jflex,…

Cette suite est ensuite transmise à l'analyseur syntaxique.

3.3 Analyse syntaxique

3.3.1 Présentation

Tout langage informatique possède un ensemble de règles définissant sa syntaxe, cet ensemble est appelé la grammaire du langage.

Une grammaire est composée de quatre éléments [17,p56]:

- l'ensemble des symboles terminaux Vt
- l'ensemble des non-terminaux : Vn
- l'ensemble des productions : P
- le symbole de départ (non-terminal) : S

Le symbole de départ est un symbole non-terminal associé à une ou plusieurs productions définissant plusieurs possibilités de décomposition du symbole de départ. Chaque décomposition comporte un ou plusieurs symboles terminaux et/ou non terminaux (un symbole terminal particulier représente la décomposition vide).

Chaque symbole non-terminal, tout comme le symbole de départ, est associé à une ou plusieurs règles de production. La particularité du symbole de départ est qu'il ne peut pas apparaître dans la décomposition d'un autre symbole.

Les symboles terminaux correspondent aux unités lexicales recherchées par l'analyse lexicale, ce sont les « briques de base » de la grammaire.

Voici un petit exemple issu de la langue française (le chapitre 1 présente des liens entre l'informatique et le monde industriel, voici ici un lien avec le monde littéraire) [16, p10]:

V_T = { il, elle, parle, est, devient, court, reste, sympa, vite }
V_N = { PHRASE, PRONOM, VERBE, COMPLEMENT, VERBETAT, VERBACTION }
S = PHRASE
P = { PHRASE → PRONOM VERBE COMPLEMENT
PRONOM → il | elle
VERBE → VERBETAT | VERBACTION
VERBETAT → est | devient | reste
VERBACTION → parle | court
COMPLEMENT → sympa | vite }

Figure 3.4 Exemple de grammaire

Dans cet exemple, le symbole de départ est S, il est décomposé en PHRASE, lui-même décomposé en d'autres symboles, eux-mêmes décomposés…

C'est un exemple très simple : un symbole non-terminal y est décomposé en une suite de symboles non-terminaux ou une suite de symboles terminaux. Bien entendu, certaines grammaires décomposent leurs symboles non terminaux en suite comportant à la fois des symboles terminaux et non terminaux.

L'analyse syntaxique consiste à déterminer si la suite de lexèmes identifiée par l'analyse lexicale respecte la grammaire du langage. Cette vérification s'effectue par dérivation, il en existe deux types : dérivation descendante ou ascendante.

3.3.2 Dérivation descendante

Cette technique est probablement la plus intuitive, elle consiste à :

- partir du symbole de départ
- sélectionner une règle de production
- remplacer le symbole non terminal par la décomposition décrite par la règle
- continuer récursivement jusqu'à ce qu'il n'y ait plus que des symboles terminaux

La décomposition du symbole de départ et les décompositions récursives suivantes peuvent être représentées sous forme d'arbre dont la racine est le symbole de départ et les nœuds descendants sont issus des décompositions.

S'il existe un arbre de dérivation dont les feuilles correspondent à la suite des lexèmes issue de l'analyse lexicale, alors cette suite est syntaxiquement correcte, elle est bien issue de la grammaire.

Exemple, le texte « Elle parle vite » est-il syntaxiquement correct ?

Le symbole de départ est S, il ne peut être décomposé qu'en PHRASE, lui-même décomposé en PRONOM VERBE COMPLEMENT.

PRONOM est décomposable en « il » ou « elle ». Le premier mot est lu « elle », le règle « PRONOM -> elle » est donc choisie, le premier lexème est donc traité.

VERBE est décomposé en VERBETAT ou VERBACTION, la lecture du lexème « parle » permet donc de sélectionner la règle « VERBE -> VERBACTION » et « VERBACTION -> parle ». Le second lexème est obtenu, l'analyseur peut donc se positionner sur le suivant.

COMPLEMENT peut être dérivé en « vite » ou « sympa », la lecture de « vite » permet de choisir la règle « COMPLEMENT -> vite ». Tous les lexèmes ont été obtenus par dérivation, le texte est donc syntaxiquement correct.

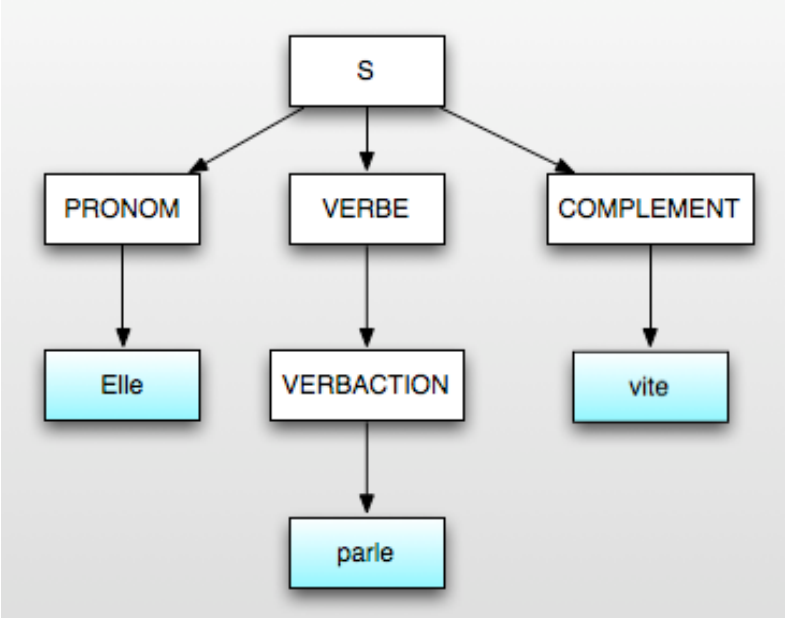


Figure 3.5 Exemple d'arbre de dérivation

Les langages définis par des grammaires de type LL sont analysés par dérivation descendante.

3.3.3 Dérivation ascendante

La dérivation ascendante consiste également à construire un arbre de dérivation afin de vérifier qu'une suite de lexèmes est syntaxiquement correcte. Mais cette dérivation se fait de bas en haut.

L'idée est de parcourir la suite de lexèmes afin d'effectuer des opérations parmi les suivantes :

- Shift : se positionner sur le lexème suivant
- Reduce : remplacer la suite de symbole enregistré depuis le dernier Reduce par un non-terminal selon une règle de production de la grammaire.

L'enregistrement des symboles se fait généralement à l'aide d'une pile. A chaque fois qu'un symbole est découvert, il est empilé. Quand l'analyseur effectue une réduction, il retire de la pile les derniers symboles empilés, ceux-ci doivent correspondre aux symboles définis dans la règle de production utilisée.

Une telle analyse peut être confrontée à deux types de conflits :

- Reduce/Reduce : Selon la grammaire et l'état de la pile et le symbole lu, l'analyseur peut choisir entre plusieurs règles pour une réduction.
- Shift/Reduce : Selon la grammaire, l'état de la pile et le symbole lu, l'analyseur peut réduire ou passer au lexème suivant.

Certains de ces conflits sont généralement résolus par l'octroi de priorité aux actions et aux productions.

Exemple, la phrase « Elle parle vite » est-elle syntaxiquement correcte ?

Le 1er lexème lu est « Elle ». Cela correspond à la partie droite de la règle «PRONOM -> Elle ». L'analyseur peut donc effectuer une réduction en remplaçant « Elle » par PRONOM. Au niveau de l'arbre, le nœud « PRONOM » est alors ajouté et défini comme parent de « Elle ». Après ces opérations, la pile contient le symbole PRONOM et l'analyseur passe au lexème suivant.

Le second lexème est « parle ». L'analyseur va donc effectuer une réduction en « VERBACTION », puis une autre réduction en de VERBACTION en VERBE. Après ces deux réductions, la pile contient PRONOM et VERBE.

Le dernier symbole lu est « vite ». Il peut être réduit en COMPLEMENT qui est empilé. La pile contient donc PRONOM VERBE COMPLEMENT. L'analyseur peut donc effectuer une réduction en PHRASE, puis une autre en S, le symbole de départ.

L'analyseur a atteint le symbole de départ en partant des symboles terminaux issus des lexèmes, le texte est donc syntaxiquement correct.

L'arbre de dérivation formé par cette analyse est identique à celui de la figure 3.5.

Les langages définis par des grammaires de type SLR, LR, LALR sont analysés par dérivation ascendante. Ces trois types d'analyses diffèrent légèrement, mais vu l'utilisation d'un générateur d'analyseur LALR et vu le souhait de ne pas décrire les détails d'implémentation, détailler les différences entre ces trois types d'analyse n'est pas nécessaire.

3.4 La représentation interne.

Comme vu au point précédent, l'analyse syntaxique produit un arbre de dérivation, aussi appelé arbre abstrait, qui indique comment le texte à analyser est obtenu à partir de la grammaire. Cet arbre est donc une représentation du texte source dans les termes de la grammaire.

Toutefois, la structure de cet arbre n'est pas la plus commode dans l'optique des traitements futurs. L'important est de stocker les éléments présents dans le texte source en utilisant les termes de la grammaire et en conservant les liens hiérarchiques qui les unissent, mais mémoriser toutes les règles intermédiaires découvertes lors de la dérivation alourdit l'arbre et n'a pas d'intérêt pour la vérification sémantique et la partie « arrière » du compilateur.

Une forme allégée est généralement utilisée, appelée « arbre syntaxique abstrait » (« abstract syntax tree » , « AST »).

Un petit exemple :

Considérons la grammaire suivante décrivant des opérations arithmétiques simples :
expression -> expression '+' terme
| expression '-' terme
| expression
terme -> terme '*' facteur
| terme '/' facteur
| facteur
facteur -> identificateur | constante
identificateur : [a-z]*
constante : 0|[1-9][0-9]

Le texte a * b – 4 correspond à la suite de lexèmes : identificateur * identificateur – constance
Cette suite correspond syntaxiquement à l'arbre syntaxique ci-dessous :

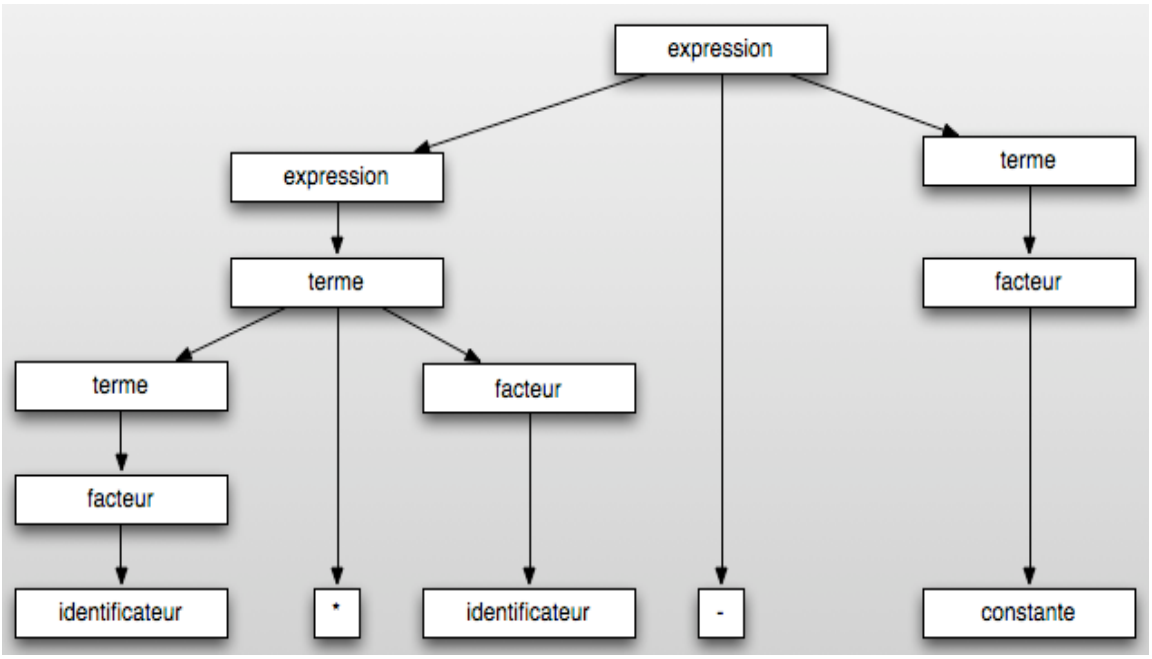


Figure 3.6 Arbre syntaxique de a * b – 4 (Identificateur * Identificateur – constante)

Quels sont les éléments du texte à exprimer sous forme de la grammaire :

- identificateur (valeur : a)
- « * »
- identificateur (valeur : b)
- « - »
- constante (valeur : 4)

Quels sont les « liens hiérarchiques » entre ces éléments ?

- Une opération de soustraction liant un symbole à deux autres symboles
- Une opération de multiplication liant un symbole à deux autres symboles

Le symbole « - » va donc être choisi comme racine de l'arbre, ses deux nœuds fils seront d'une part un sous-arbre dont la racine est le symbole « * » et d'autre part la constante.

Les autres informations présentes dans l'arbre de dérivation, par exemple, que l'identificateur de valeur « a » est obtenu par l'intermédiaire d'un facteur étant lui-même un terme, est sans intérêt pour la vérification sémantique et la partie arrière du compilateur, puisque celle-ci s'intéresse à la sémantique des symboles rencontrés et non à la structure exacte de sa syntaxe.

L'arbre syntaxique abstrait est le suivant :

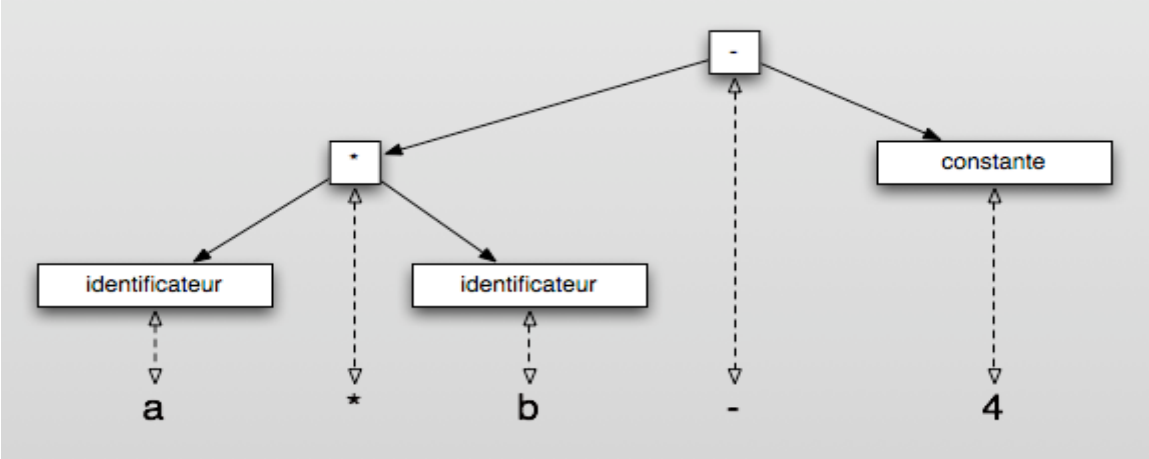


Figure 3.7 Arbre Syntaxique Abstrait de a * b - 4

Déterminer les éléments à inclure dans cet arbre est trivial, il s'agit des lexèmes. Par contre, reclasser ces éléments hiérarchiquement est plus compliqué, cela nécessite des connaissances au niveau de la sémantique. En effet, comment l'analyseur pourrait-il, à partir de la grammaire, déterminer que le « * » doit être la racine du sous-arbre de gauche ? Ce problème n'est pas trivial et devient de plus en plus compliquée selon la complexité de la grammaire et de la sémantique sous-jacente. Cette opération est généralement laissée au programmeur : celui-ci va fournir la grammaire au générateur d'analyseur syntaxique, mais aussi des règles strictes, souvent sous forme de code dans un langage de programmation, permettant à l'analyseur de construire l'arbre syntaxique abstrait sans ambiguïté.

Par exemple, les « actions » (lignes de codes) définies dans les grammaires fournies aux générateurs Yacc, Bison ou CUP [30].

3.5 Analyse sémantique

Après les analyses lexicale et sémantique, le texte source a été traduit en une représentation interne, généralement sous forme d'AST. Le texte est syntaxiquement correct, mais est-il correct d'un point de vue sémantique ?

Par analogie à la langue française, l'exemple de la grammaire 3.4 est intéressant. Le texte « Elle parle vite » est syntaxiquement correct, ceci est démontré par l'arbre de dérivation de la figure 3.5. Si la sémantique du langage de cette grammaire est issue de la sémantique de la langue française, ce texte est sémantiquement correct.

Le texte « Elle parle sympa » est lui aussi syntaxiquement correct, tout comme l'exemple précédent, il correspond aux lexèmes PRONOM VERBE COMPLEMENT. Son arbre de dérivation est donc identique à celui de la figure 3.5, hormis la dérivation de COMPLEMENT en « sympa » et non en « vite ».

Mais ce texte est-il sémantiquement correct ? A l'exception peut-être de versions particulières du français pratiquées dans de lointaines contrées, « Elle parle sympa » n'est pas sémantiquement correcte.

Dans le domaine des langages informatiques plus conventionnels comme C, Pascal, l'analyse sémantique s'intéresse principalement à ces problèmes :

- **La portée des identificateurs**
  Lorsqu'une variable est utilisée dans un bloc, a-t-elle été déclarée correctement au préalable ? Si elle a été déclarée, cette déclaration est-elle à prendre en compte dans le bloc de code courant ? Si une variable est déclarée dans un bloc avec un nom identique à une variable d'un autre bloc, que se passe-t-il ? Ces questions ne peuvent pas être résolues par la grammaire seule.

- **Le typage**
  Considérons un langage permettant d'effectuer des opérations arithmétiques et logiques sur des variables de type entier, réel et booléen, un sous-ensemble de la grammaire serait comme ceci :

  facteur -> facteur '+' terme
  terme -> identificateur | constante

  Si « identificateur » représente les variables de ce langage, et « constante » représente les constantes de tout type, alors la grammaire autorise à effectuer 5 + true. Ce n'est pas correct sémantiquement. L'analyse sémantique va effectuer cette vérification. Dans ce cas simple, une grammaire utilisant des opérateurs différents pour les additions de types et un système de nommage différent pour les différents types de variables permettrait de vérifier les contraintes de typage. Mais cette grammaire serait complexe, et à l'échelle d'un véritable langage, elle deviendrait vite ingérable.

De façon plus générale, l'analyse sémantique est chargée de vérifier les contraintes non gérées par la grammaire. Ces contraintes et la façon de les vérifier dépendent du langage et de sa sémantique.

4. Présentation de TVL

Comme vu dans les chapitres précédents, les Feature diagrams représente un outil intéressant en vue de la modélisation de la variabilité. Plusieurs versions de features diagrams existent, mais sont généralement sous forme graphique.

Ce chapitre est principalement inspiré des articles de références de TVL : « A Text-based Approach to feature Modelling » [19] et « Syntax and Semantics of TVL [20]».

TVL est l'abréviation de Textual Variability Language. Il s'agit donc d'un langage permettant de représenter la variabilité, mais sous forme textuelle.

Les articles de références de TVL [19],[20] présentent les avantages d'une notation textuelle par rapport à une notation graphique.

Même si un schéma graphique pourrait sembler plus simple à un utilisateur non initié, une utilisation à grande échelle serait probablement compliquée, vu la lourdeur du schéma.

Un format textuel est plus compact. De plus il peut être traité comme un texte et comme un langage classique.

En effet, la réalisation d'outils d'encodage avec fonctions d'auto-completion, de correction syntaxique et de recherche est plus aisée au format textuel.

Le parsing du code, sa validation et une étude de faisabilité des modèles sont également plus aisés puisqu'il s'agit d'un langage classique pour lequel des techniques et outils d'analyse lexicale, syntaxique et sémantique sont éprouvés, diversifiés et libres d'accès.

Une définition complète de la grammaire de TVL ainsi que de sa sémantique, dans leur version antérieure aux modifications apportées par ce mémoire, sont décrites dans les articles de références de TVL [19] et [20].

La première section introduit brièvement les fondements théoriques de TVL, les notions de syntaxe abstraite, syntaxe concrète ainsi que les trois éléments permettant de définir sa sémantique (domaine syntaxique, domaine sémantique et fonction sémantique) y seront présentés. Ces éléments théoriques seront ensuite utilisés dans la suite du chapitre.

La seconde section présente les principales fonctionnalités du langage TVL en les associant aux concepts : leurs concepts équivalents dans les Feature Diagrams.

Enfin, une brève description de l'implémentation de TVL, antérieure à ce mémoire, est présentée dans la dernière section, basée sur le mémoire de P. Faber [24] ainsi que sur une rétro-analyse du code source. Cette description utilisera les concepts d'analyses de langages vus au chapitre précédent.

4.1 Définitions de TVL

Le chapitre précédent introduit la notion de grammaire, la syntaxe de TVL est définie par une grammaire de type LALR, présentée en détail dans l'article « Syntax & semantic of TVL » [20] (p.4-7). La section 4.1.2 décrit la syntaxe de TVL par le biais d'exemples, en illustrant la correspondance avec les feature diagrams.

La sémantique de TVL est également définie formellement. Cette section, basée sur l'article « Syntax & Semantic of TVL » [20] (p. 7-10), vise à présenter brièvement les étapes et les concepts de la définition de cette sémantique.

Cette définition respecte les recommandations de Harel et Rumpe [18], elle consiste donc en trois éléments :

- **Une syntaxe formelle ($\mathcal{L}$)**: elle représente l'ensemble des règles syntaxiques permettant de définir tous les modèles (ici modèle TVL) valides (le domaine syntaxique) .
- **Un domaine sémantique ($\mathcal{S}$)**: il s'agit des concepts réels représentés par les modèles TVL.
- **Une fonction sémantique ($\mathcal{M}: \mathcal{L} \rightarrow \mathcal{S}$)**: elle permet de définir la signification d'un modèle, elle associe les éléments d'un modèle à leur signification dans le monde réel.

La syntaxe formelle utilisée dans la sémantique n'est pas la grammaire TVL. Il s'agit d'une extension du langage « Free Feature Diagrams » (ou FFD) de Schobbens et al ; elle est souvent notée ($\mathcal{L}_{\mathbf{TVL}}$).

En effet, la grammaire de TVL définit la syntaxe concrète de TVL, ce que l'utilisateur peut écrire dans un modèle TVL. Cette syntaxe concrète peut être traduite en une syntaxe abstraite ($\mathcal{L}_{\mathbf{TVL}}$), utilisée dans la définition de la sémantique comme syntaxe formelle.

Cette traduction de la syntaxe concrète en syntaxe abstraite est décrite dans l'article Syntax & semantic of TVL [20].

Elle consiste dans un premier temps à exprimer les éléments de TVL dans un sous-ensemble d'éléments de TVL. Ce sous-ensemble comporte les éléments de base de TVL. Les autres éléments de TVL ont toujours une équivalence dans le sous-ensemble de base, ils permettent une plus grande facilité d'utilisation de TVL, mais n'apportent rien de plus au niveau de sa signification. Le langage composé de ce sous-ensemble du langage TVL est appelé forme normale de TVL (TVL_{NF}).

La seconde phase de traduction de la syntaxe de TVL consiste alors à exprimer les éléments de TVL_{NF} en $\mathcal{L}_{\mathbf{TVL}}$.

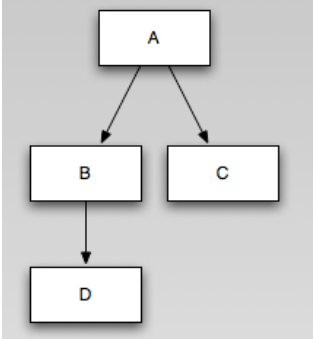
Petite remarque concernant l'illustration de la sémantique : puisque cette sémantique est définie avec un langage abstrait correspondant au langage concret TVL, les illustrations de cas pratiques doivent se faire via des feature diagrams respectant cette sémantique.

4.2 Description des composants de TVL

4.2.1 Forme générale d'une hiérarchie de Feature

Une hiérarchie de feature est aussi appelée un modèle, elle correspond au graphe d'un feature diagram.

Le point 2.3 présente les composants de base d'un feature diagram : les nodes, les liens et le graphe formé par ceux-ci, exemple :



Dans cet exemple, la racine est le feature A. Celui-ci possède deux sous-features (enfants): B et C. Il s'agit d'une décomposition « AND », B et C doivent être présents.

TVL fournit des concepts équivalents, un feature est représenté par son nom et peut posséder un corps (entre { }).

Ce corps définit les sous-features en utilisant le mot-clé « group », suivi du type de décomposition. La traduction de l'exemple en TVL est donc :

```
root A {
      group allOf{
                  B  group allOf { D },
                  C
      }}
```

TVL fournit un mot-clé indiquant les features optionnels. Dans ce cas, lorsque l'on déclare les sous-features membres de la décomposition d'un feature, le mot clé « opt» est placé devant les noms de sous-features optionnels.

«root » indique la racine (une seule racine par hiérarchie), « group » suivi de « allOf » indique qu'il s'agit d'une décomposition « AND ».

Comme pour les feature diagrams, quatre types de décompositions existent et sont comparables aux types de décompositions des feature diagrams (cfr 2.4)

Décompostion de Feature Diagram	**Décomposition de TVL**
AND	allOf
OR	someOf
XOR	oneOf
Basée sur cardinalités	[min..max]

La section 4.1.2 détaille les possibilités de ces décompositions.

Outre la définition de la hiérarchie, le corps d'un feature peut contenir des attributs (cfr 4.2.3) et des contraintes (cfr 4.2.4), de façon similaire aux attributs (cfr 2.6) et contraintes (cfr 2.5) des feature diagrams.

Plusieurs syntaxes de définition de la hiérarchie existent. Notamment le « split up » décrit dans l'article « A Text-based approah to feature Modelling" » [19](p7). Cette syntaxe consiste à ne pas définir le corps d'un feature lors de sa déclaration dans le corps de son parent, mais de définir son corps séparément.

Le « split up » est une application du principe d'extension : Le corps d'un feature peut être défini en plusieurs étapes, sa décomposition ne peut être définie qu'une seule fois et ne peut pas être étendue par la suite, mais une définition d'attributs et de contraintes supplémentaires est toujours possible, bien qu'en abuser ne soit pas très recommandé au niveau de la lisibilité.

Un autre « sucre syntaxique » concerne les corps de feature ne possédant rien d'autre que leur décomposition, le « group » peut alors suivre directement le nom du feature, comme indiqué dans « A Text-based approah to feature Modelling» [19](p7).

L'application de ces deux possibilités syntaxiques permet d'écrire l'exemple de la façon suivante :

```
root A {
      group allOf { B, C }
}

B group allOf{ D }
```

Par exemple, d'autres attributs ou contraintes pourraient être ajoutés par une nouvelle extension de la définition du corps de B (cfr 4.2.3 pour description des attributs de TVL) :

```
B {
      int valeur ;
      int code ;
}
```

Un mécanisme d'inclusion, défini dans « A Text-based approah to feature Modelling" » [19] (p9), existe également en TVL. Il consiste à inclure en un point donné d'un fichier TVL le code défini dans un autre fichier TVL. Une application typique de ce mécanisme consiste à décrire la hiérarchie dans le fichier principal et d'utiliser des « include» pour référencer d'autres fichiers TVL contenant les détails.

L'exemple précédent serait donc composé de deux fichiers :

structure.tvl contenant :

```
root A {
      group alllOf { B, C }
}
Include (detail.tvl)
```

et detail.tvl contenant :

```
B group allOf{ D }
```

4.2.2 Types de décomposition

Comme cité dans le point 4.2.1, les types de décomposition permis par TVL correspondent aux décompositions généralement permises par les feature diagrams (2.4).

Petit rappel :

Décompostion de Feature Diagram	Décomposition de TVL
AND	allOf
OR	someOf
XOR	oneOf
Basée sur cardinalités	[min..max]

Tout comme dans le cas des feature diagrams, la décomposition par cardinalités de TVL est aussi une généralisation des 3 autres types de décomposition :

Décompostion booléenne	Décomposition par cardinalités
allOf	[N..N]
someOf	[1..N]
oneOf	[1..1]

Rem : N représente le nombre de sous-feature du feature parent de la décomposition

Toutefois, cette section va plus loin concernant certains cas d'implémentation de ces décompositions, notamment l'influence du caractère optionnel des features.

L'implémentation de TVL définit le mot clé « OPT » comme étant prioritaire sur les décompositions, voyons ci-dessous les conséquences relatives aux quatre différents types.

- **Décomposition allOf**

  Considérons le cas suivant :

```
root R group allOf{
      opt A,
      B
}
```

  Le allOf signifie que tous les sous-features de « R » doivent être présents. Or, « A » est optionnel. Il y a donc un conflit entre « allOf » et « opt ». TVL résout ce problème en accordant la priorité à « opt ».

  Les cas suivants seront acceptés :
  - Le produit contient A et B : « allOf » et « opt » sont tous deux respectés.
  - Le produit contient uniquement « B » : à première vue, « allOf » n'est pas respecté puisque que « A » est absent, mais le caractère optionnel de « A » étant prédominant, ce produit est accepté.

  Par contre, un produit contenant uniquement « A » ne sera pas accepté, car « B » n'est pas optionnel. Bien entendu, un produit ne contenant ni « A » ni « B » sera également rejeté.

- **Décomposition someOf**

  Soit le cas :

```
root R group someOf{
      opt A,
      B
}
```

  Le « someOf » signifie qu'au moins un feature parmi « A » et « B » doit être présent. La priorité accordée au caractère optionnel de « A » modifie l'effet de cette contrainte :

  Voici les cas acceptés :

  - « A » et « B » sont présents : Les contraintes « someOf » et « opt » sont toutes les deux vérifiées.
  - Seul un des deux feature parmi « A » et « B » est présent : les contraintes « someOf » et « opt » sont toutes les deux vérifiées.
  - Ni « A », ni « B » n'est sélectionné : apparemment, la définition de someOf (A ou B) n'est pas respectée, mais le caractère optionnel de « A » étant prioritaire, ce cas est accepté.

- **Décomposition oneOf**

  Soit le cas :

```
root R group oneOf{
      opt A,
      B
}
```

  Le « oneOf » signifie qu'un produit est valide si « A » ou « B » est sélectionné (mais pas les deux). La prépondérance accordée au caractère optionnel de « A » a également une influence sur la validité de ce produit.

  Voici les cas acceptés :
  - Seul un des deux features parmi « A » et « B » est sélectionné : ce cas est accepté.
  - Ni « A », ni « B » n'est sélectionné : ce cas est similaire à la même configuration d'une décomposition « someOf ».

  Par contre, si « A » et « B » sont sélectionnés, ce cas est refusé, car « oneOf » interdit de sélectionner les deux features. Le caractère optionnel de « A » n'y change rien.

- **Décomposition basée sur les cardinalités**

  Soit le cas :

```
root R group [2..3] {
      opt A,
      B,
      C
}
```

  Les cardinalités [2..3] signifient qu'un produit doit sélectionner au moins deux et au maximum trois sous-features parmi les features « A », « B », « C ».

  Toutefois, le caractère optionnel de « A » influence les choix possibles.
  Voici les cas acceptés :
  - « A », « B » et « C » sont sélectionnés : les cardinalités sont respectées.
  - 2 features parmi « A », « B », « C » sont sélectionnés : les cardinalités sont respectées.
  - 1 seul feature est sélectionné parmi « B » ou « C » : même si le nombre de sous-features est inférieur à la cardinalité minimale, le caractère optionnel de « A » permet de ne pas sélectionner « A », sans que son absence ait une influence sur le respect des cardinalités.

  Par contre, un produit ne contenant que « A » n'est pas accepté.

Intuitivement, en exprimant les décompositions allOf, someOf et oneOf en termes de cardinalités, les cas précédents permettent l'observation suivante :

La cardinalité minimale est diminuée du nombre de sous-features optionnels qui ne sont pas sélectionnés dans le produit.

La sémantique de TVL, définie dans les articles de références de TVL [19,20] permet d'expliquer et de formaliser cette observation…

Voici quelques éléments issus de la syntaxe abstraite de TVL :

- F : ensemble de features
- $r \in F$: racine
- $DE \subseteq F \times F$: la relation de décomposition entre les features.
  Par exemple, dans le couple $(p,e) \in DE$, noté aussi $p \rightarrow e$, p est le feature parent et e l'enfant de p.
- $\lambda : F \rightarrow N \times N$: fonction indiquant les cardinalités de décomposition d'un feature.
  Par exemple, $\lambda(f) = [m..n]$ signifie que la cardinalité de décomposition du feature f est m, et la maximale est n.
- $\omega : F \rightarrow \{0, 1\}$: fonction indiquant si un feature est optionnel ou non.
  Par exemple, $\omega(f) = 0$ signifie que le feature f est optionnel

Une des règles exprimées dans la définition de la fonction sémantique de TVL indique la signification des cardinalités de décomposition (cfr « A text-based approach to feature modelling » [19], p15), en voici un bref rappel.

Soit c un ensemble de features valides, en d'autres termes, un produit valide.

Pour tout feature f appartenant à ce produit c, dont les cardinalités de décomposition sont [m..n], ce qui se note : $\forall f \in c \;\; \lambda(f) = \langle m..n \rangle$

Les cardinalités de décomposition sont respectées si les conditions suivantes le sont :

(1) $m - | opt_F | \leq | mand_c |$
(2) $| all_c | \leq n$

Avec :

$Opt_F = \{g \mid g \in F \wedge \omega(g) = 1 \wedge f \rightarrow g\}$
(l'ensemble des features optionnels descendant du feature f)

$mand_c = \{g \mid g \in c \wedge \omega(g) = 0 \wedge f \rightarrow g\}$
(l'ensemble des features obligatoires descendants de f et sélectionnés dans c)

$all_c = \{g \mid g \in c \wedge f \rightarrow g\}$
(l'ensemble des features descendants de f, sélectionnés dans le produit c)

Voyons comment ces éléments théoriques peuvent justifier l'observation relative à la décrémentation de la cardinalité minimale.

Dans le cas suivant :

```
root R group [2..3] {
      opt A,
      B,
      C
}
```

$\lambda(R) = \langle 2..3 \rangle$
$opt_F = \{A\}$

Si le produit c, contient les features suivants : {R, B}, alors
$mand_c = \{B\}$ et $all_c = \{B\}$

Les cardinalités de décomposition de c sont bien respectées, en effet :

$m - | opt_F | \leq | mand_c |$ devient $2 - 1 \leq 1$, ok
$| all_c | \leq n$ devient $1 \leq 3$, ok

Par contre, si le produit c contient les features suivants : {R, A}, alors
$mand_c = \{\}$ et $all_c = \{A\}$

Les cardinalités ne sont pas respectées :

$m - | opt_F | \leq | mand_c$ devient $2 - 1 \leq 0$, FAUX !
$| all_c | \leq n$ devient $1 \leq 3$, ok

Ces exemples illustrent bien que la cardinalité minimale est décrémentée du nombre de features optionnels dans N, mais que le nombre de ces features optionnels présents dans le produit a aussi une importance.

En effet, si f est le feature parent de la décompostion, la règle (1) vérifie que le nombre de features obligatoires descendants de f et sélectionnés dans le produit est supérieur ou égal à la nouvelle valeur de la borne inférieure.

Ce qui implique que pour tout feature optionnel dans N, on décrémente de 1 la borne inférieure, mais on décrémente également la valeur à comparer avec cette borne inférieure si ce feature est sélectionné. Or, décrémenter les deux termes d'une égalité de la même valeur équivaut à ne rien décrémenter. Ce qui équivaut à dire que l'on décrémente la borne inférieure du nombre de features optionnels non sélectionnés.

4.2.3 Attributs

TVL supporte 4 types d'attributs :

- types primitifs : nombres entiers (int), nombres réels (real), booléen(bool)
- un type enumération (enum)

L'utilisateur peut redéfinir de nouveaux types, simples ou composés, basées sur ces types. Par exemple : soit un feature « MotherBoard » utilisant un attribut de type « dimension », composé de deux attributs de types int.

```
struct dimension {
      int height;
      int width;
}
Motherboard {
      dimension size;
}
```

La version actuelle de TVL ne permet pas d'inclure dans un type d'attribut composé, un sous-attribut lui-même de type composé.

TVL permet la définition de contrainte portant sur les valeurs de l'attribut. Voici un exemple des 4 formes de déclaration d'un attribut (cfr «A text-based approach to Feature modelling » [19], p8).

```
Feature {
   int x;
}
```
(a) Basic

```
Feature {
   int x in [0..10];
}
```
(b) Domain restriction

```
Feature {
   int x is 10;
}
```
(c) Fixed value

```
Feature {
   int x, ifIn:  is 10,
          ifOut: is  0;
}
```
(d) Conditional value

Figure 4.1 Différentes declarations d'un attribut (extrait de[19], p8)

Comme l'indique ces exemples, un attribut, outre la déclaration de son type, peut aussi recevoir une contrainte de valeur de domaine (b), une valeur imposée et non modifiable (c). La contrainte de domaine ou la valeur imposée peuvent être différentes selon que le feature soit sélectionné dans le produit ou non.

La grammaire de TVL (cfr « Syntax & Semantic of TVL [20], p6) indique les différents cas d'utilisation de ces formes, y compris la manière de les combiner.

4.2.4 Contraintes

Les contraintes généralement utilisées par les feature diagrams (cfr 2.5) existent dans TVL (cfr articles de références de TVL [19] (p8) et [20] (p4, 6-7)).

TVL fournit les mots-clés « requires » et « excludes », mais aussi la possibilité d'exprimer des contraintes sous forme d'expressions booléennes, contenant des opérateurs booléens et arithmétiques, des valeurs appartenant aux types présents dans TVL, des identificateurs de variables, de features et des constantes.

«Syntax & Semantic of TVL » [20] (p4,7) détaille les différentes possibilités de ces expressions.

Ce chapitre ne détaille pas les contraintes. Les modifications apportées à TVL dans le cadre de ce mémoire, présentées aux chapitres 5, impliquent une révision des contraintes de TVL, aux niveaux syntaxique et sémantique. Le chapitre 6 proposera une nouvelle version de la syntaxe et sémantique des contraintes de TVL, avec les rappels nécessaires de certaines caractéristiques des contraintes antérieures à ce mémoire.

Dans la suite de ce mémoires, ces contraintes seront nommées : contraintes « cross-tree », afin de différencier ces contraintes, pouvant porter sur des éléments à travers l'arbre composé par les éléments d'un produit, du terme contrainte au sens large représentant les contraintes « cross-tree » mais aussi les contraintes telles que le respect de la décomposition et le respect des types et domaines des d'attributs.

4.2.5 Règles de validité d'un fichier TVL

Comme c'est également le cas de langage de programmation comparable au C, un code source TVL doit vérifier certaines règles qui ne peuvent être garanties par la grammaire seule.

Ces règles sont définies en détail dans « Syntax é Semantic of TVL » [20] p8. Voici une brève présentation des principales règles.

- **Nommage, portée et référence**

  Les règles de nommage sont assez proches de celles en vigueur dans d'autres langages, tel que le C.

  Une particularité de TVL est l'obligation de commencer les noms de features par une majuscule, alors que les noms d'attributs et de types doivent commencer par une lettre en minuscule.

  Un nom doit être unique à l'intérieur de son contexte, ce qui implique que les noms des éléments suivants doivent être distincts :
  - sous-features et attributs d'un même feature
  - tous les types déclarés
  - les constantes
  - attributs membres d'une même structure
  - valeurs des énumérations par énumération

De plus,

- Une valeur d'énumération doit être différente des noms d'attributs, de features ou de constantes.
- Une constante ne peut avoir le même nom qu'un attribut.

TVL permet l'utilisation de noms qualifiés. Par exemple, un feature « R » peut être composé de features « A » et « B », tous deux composés d'un feature « X » distinct de son « cousin ». Le nom qualifié du sous-feature de A appelé X est donc « A.X », alors que le fils de B est « B.X ».

Ceci est particulièrement utile lors de la définition de contraintes ou la définition du corps d'un feature selon la syntaxe « split up » (cfr 4.2.1).

Afin de faciliter la référence à un feature, les mots clés « root », « parent » et « this » sont disponibles et permettent de référencer respectivement la racine, le parent du feature courant et le feature lui-même.

Le mot clé « children » permet de référencer la collection des enfants d'un feature.

- **Correction des types**

TVL est fortement typé et ne permet pas le casting de type. Ce n'est pas un langage orienté objet, il ne comporte donc aucune notion d'héritage ni de polymorphisme.

La vérification des types peut donc se faire de façon statique et est comparable à celle du langage C, cette vérification se fait à plusieurs niveaux :

- affectation de variables : le type de la valeur doit correspondre au type de la variable.
- opérateurs : les composants d'un opérateur booléen doivent être booléens, les composants d'un opérateur arithmétique doivent être entiers ou réels.
- Les contraintes doivent être des expressions de type booléen, même si les sous-expressions qui la composent utilisent des expressions arithmétiques.
- Les ensembles définis en extension doivent contenir des éléments de même type et compatibles avec le type de l'ensemble. Par exemple, un ensemble de type énumération ne peut contenir que des valeurs d'énumérations, pas de valeurs numériques, ni booléens ni d'autres types.

- **Autres règles : Structure de la hiérarchie et cardinalités**

  - Tout comme les feature diagrams, la relation de décomposition des features doit être acyclique.
  - Une seule racine existe.
  - Un feature ne peut comporter qu'une seule décomposition.
  - Une cardinalité minimale doit être inférieure ou égale à la cardinalité maximale associée.
  - Les cardinalités minimale et maximale d'une décomposition doivent être compatibles avec le nombre de sous-feature définis.

4.2.6 Feature partagé

Un feature diagram est, pour rappel (cfr 2.3.3), un graphe orienté, acyclique et possède une seule racine.

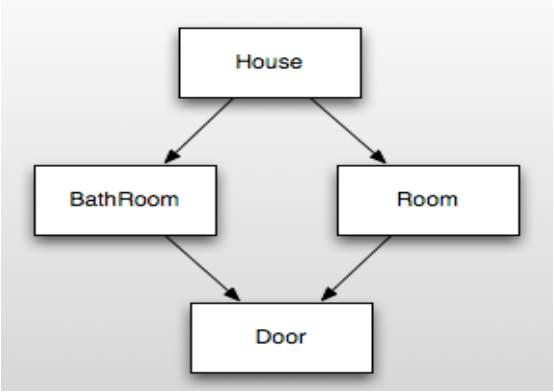
La structure hiérarchique de TVL, dans laquelle un feature parent peut posséder une décomposition en sous-features, garantit le caractère orienté (cfr 4.2.1).

Les règles présentées au point précédent (4.2.5) garantissent le caractère acyclique et l'unicité de la racine.

Le graphe d'un feature diagram est souvent un arbre, c'est également le cas d'un modèle TVL. Toutefois, un feature TVL peut avoir plusieurs parents, c'est alors un feature partagé entre ses plusieurs parents.

TVL permet cela via l'utilisation du mot-clé « shared » (cfr « Syntax & Semantic of TVL [20], p3). Exemple : soit une feature « House », composé de features « BathRoom » et « Room », tous deux partageant un même feature « Door ».

Voici le feature diagram et le code TVL de cet exemple :



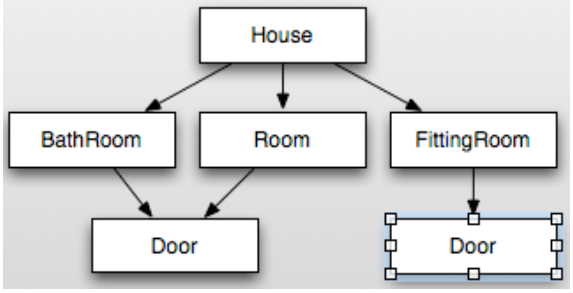

```
root House group allOf{
      BathRoom group allOf
            { Door},
      Room group allOf
            { shared Door}
}
```

Un des features parents du feature partagé n'utilise pas le mot-clé « shared », tous les autres doivent l'utiliser.

Selon la section 4.2.5 décrivant les règles de nommage, un nom de feature est unique sous son feature parent, un modèle contenant plusieurs feature « Door » serait donc possible.

Mais que se passe-t-il si certains de ces feature « Door » sont partagés et d'autres pas ?

Voici un exemple sous forme de feature diagram avec le code TVL associé :



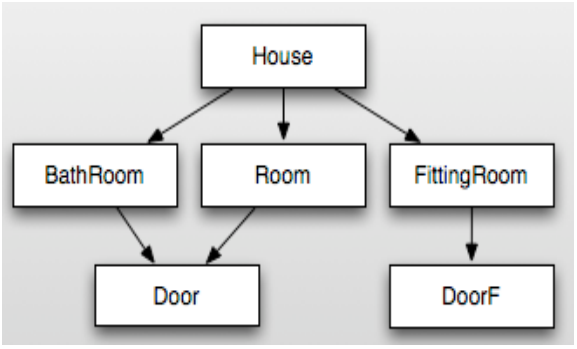

```
root House group allOf{
      BathRoom group allOf
            { Door},
      FittingRoom group allOf
            { Door},
      Room group allOf
            {shared Door}
}
```

Dans cet exemple, comment identifier selon le code TVL à quel « Door » correspond le « Door » sous « Room » (noté « Room.Door ») ? Est-ce « BathRoom.Door » ou « FittingRoom.Door » ? La syntaxe de TVL ne le permet pas. Afin d'éviter ce problème, TVL impose que le nom d'un feature partagé soit unique dans tout le modèle TVL.

L'exemple ci dessus n'est pas permis par TVL, puisqu'un feature nommé « Door » est partagé et qu'un **autre** feature **«** Door » existe également.

Sans l'ajout de cette contrainte, la feature « Door » partagé correspondant à « Room.Door » aurait dû être spécifié, par exemple par une notation de la forme « shared Door with BathRoom », permettant en quelque sorte d'ajouter un représentation des flèches du feature diagram.

Cela permettrait d'éviter une contrainte de nommage, mais alourdirait la syntaxe du langage, sans apporter réellement un plus au niveau fonctionnel, puisqu'il suffirait de renommer un des feature « Door » par un autre nom. Le feature diagram et le code TVL seraient donc les suivants :




```
root House group allOf{
      BathRoom group allOf
            { Door},
      FittingRoom group allOf
            { DoorF},
      Room group allOf
            { shared Door}
}
```

Ce code TVL permet donc d'associer le shared "Door" sous « Room » au "Door" sous « BathRoom » puisqu'il n'y a plus qu'un seul "Door". Cet exemple est donc valide.

4.2.7 Exemple récapitulatif.

Voici un fichier TVL extrait de l'article [19] (p6). Ce fichier contient un exemple des principaux éléments décrit dans le chapitre 4 : hiérarchie de features avec décompositions, définitions d'attributs avec restrictions de domaine, de contraintes, définition du corps d'un feature en extension (split up), définition de type simple, définition de type composé.

```
//Declaring a custom type;
enum orientation in {horizontalLeft, horizontalRight, vertical};

//Declaring a structured type;
struct coord {
    int x;
    int y;
}

//The feature model
root Document{
    //And-decomposition of the root feature :
    group allOf {
        Sheet group [*..*] {
            opt Tab, //an optional feature
            Page,
            opt Hole,
            Media,
            Staple,
            NumberingMethod
        },
        opt SpineCaption
    }

    //Attribute declarations of the root feature:
    enum type in {normal, booklet, perfectBinding};
    enum stackMethod in {none, offset, mixed};

    //A contraint :
    Document.type == booklet -> !Sheet.Hole;
}

//The features and Holes are extended
SpineCaption {
    orientation orient;
    ifIn : Document.type in {booklet, perfectBinding};
}

Hole {
    coord position;
}
```

FIGURE 4.2 Modèle TVL extrait du Feature Model PRISMAprepare (extrait de [19], p6)

4.3 Architecture et fonctionnement général du parser TVL

Le parseur TVL est implémenté de manière relativement traditionnelle en ce qui concerne la partie avant. Ce parseur n'étant pas un compilateur, la partie arrière est plus spécifique. Cette partie arrière consiste à transformer la représentation interne générée par la partie avant afin d'effectuer une étude de faisabilité du modèle.

Voici un schéma décrivant le fonctionnement de la partie avant dans les termes du chapitre 3.

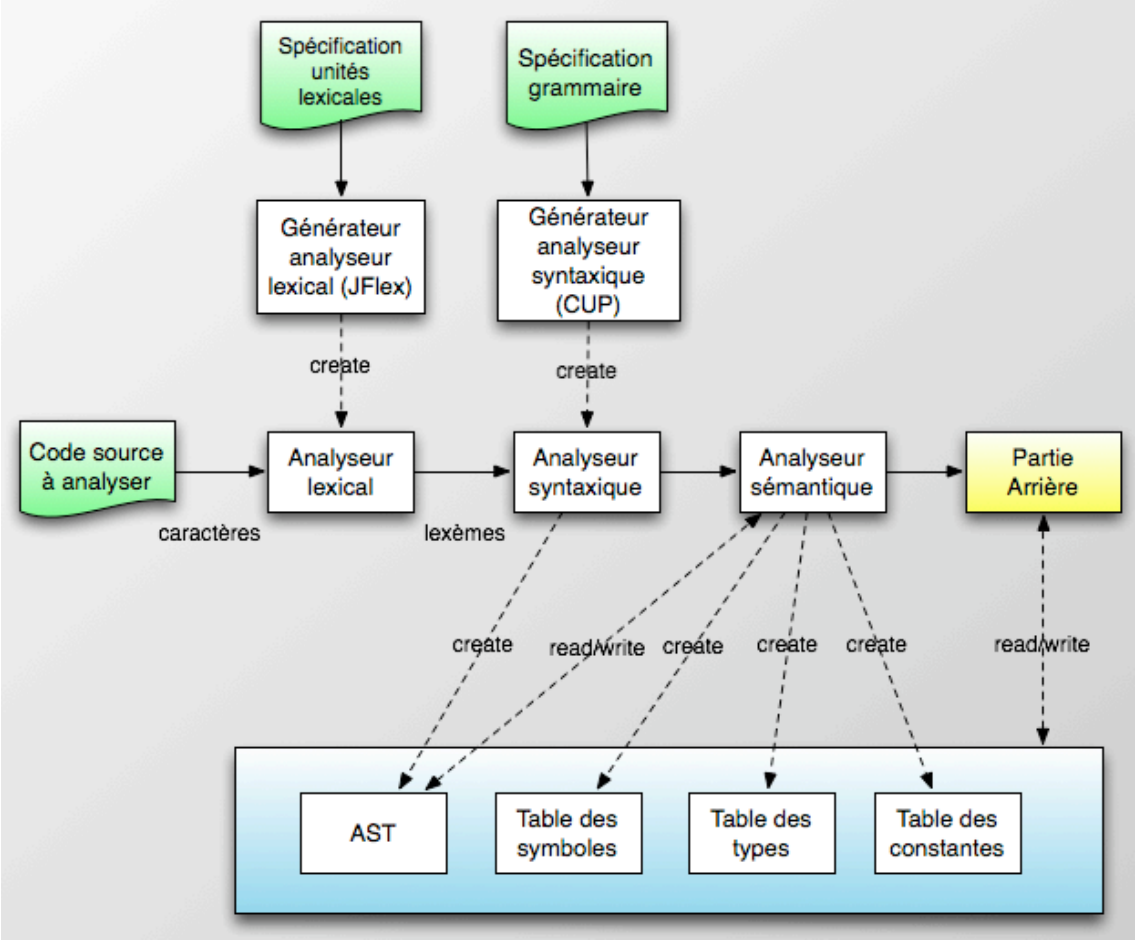


FIGURE 4.3 Structure du parseur TVL

L'analyse d'un modèle TVL s'effectue en 3 phases :

- analyse lexicale
- analyse syntaxique
- analyse sémantique

L'analyse lexicale est réalisée par un analyseur lexical (cfr 3.2) généré par un générateur d'analyseur lexical. L'outil choisi est JFlex [29], il utilise un fichier de définition des unités lexicales afin de générer l'analyseur.

Ensuite, l'analyse syntaxique (cfr 3.3) est réalisée par un analyseur syntaxique généré par CUP [30]. Ce générateur permet de générer des analyseurs syntaxiques ascendants (cfr 3.3.2), construits à partir d'une grammaire LALR (cfr Grune et al. [15], p150).

L'analyseur syntaxique crée une représentation interne du fichier analysé sous forme d'un arbre syntaxique abstrait (cfr 3.4). Des actions incluses dans la définition de la grammaire permettent à CUP de générer le code nécessaire dans l'analyseur afin de créer l'AST.

L'AST est ensuite utilisé par l'analyseur sémantique, écrit « à la main ». Cet analyseur va vérifier les contraintes non vérifiables par les analyses lexicale et syntaxique :(cfr 3.5, 4.2.5).

Il va également générer 3 structures de données supplémentaires, décrites en détail dans le mémoire de P. Faber ([24], chap. 6):

- **Table des symboles**
  Cette table va contenir les informations relatives aux features : leur nom, décomposition, attributs et contraintes.

- **Table des types**
  Cette table contient tous les types déclarés dans le modèle.

- **Table des constantes**
  Cette table contient la description de toutes les constantes définies dans le modèle : le nom, le type et la valeur de chaque constante.

Les chapitres suivants de ce mémoire utilisant ces structures de données détailleront plus en détails les aspects de leur implémentation nécessaire à ce mémoire.

4.4 Conclusion

Ce chapitre a présenté la syntaxe et la sémantique de TVL. Les fonctionnalités de ce langage s'avèrent très proches des feature diagrams ; la principale différence est son format textuel.

Mais une autre différence remarquable existe, TVL ne permet pas l'utilisation de cardinalité de feature : une instance d'un feature ne peut apparaître qu'une seule fois dans un produit. Le chapitre suivant vise à introduire cette notion de « clonage » dans TVL.

Partie 2 : Contribution

5. Cardinalités de Feature

Ce chapitre vise dans un premier temps à présenter les motivations de l'ajout de cardinalités de feature à TVL. La sémantique de ces cardinalités, définie dans l'article de référence de Michel et al. [22], sera ensuite présentée afin d'expliquer la notion de clonage au sens de TVL ainsi que les conséquences de l'introduction de clonage dans TVL.

Ce chapitre décrira les adaptations de la sémantique à propos des cardinalités de feature et des cardinalités de groupe, ainsi que différents cas où un feature peut être omis (optionalité du feature). Une définition formelle de la sémantique sera présentée à la suite de ces explications.

Ensuite, les éléments définis dans cette sémantique, sous forme de syntaxe abstraite, seront transposés à TVL, sous forme de syntaxe concrète et de règles sémantiques à vérifier lors de l'analyse sémantique d'un modèle TVL.

Ce cinquième chapitre décrira alors les adaptations à apporter à la grammaire de TVL afin d'y définir précisément les modifications de la syntaxe concrète, à l'exception des adaptations du sous-ensemble de TVL permettant d'exprimer les contraintes « cross-tree ». En effet, ces contraintes « cross-tree » seront traitées par le chapitre suivant.

Enfin, il décrira les modifications de l'implémentation de TVL, au niveau des différentes étapes de l'analyse du langage, conformément aux étapes définies dans le chapitre 3 et à l'architecture de TVL définie au chapitre 4.

Remarque **:** Ce chapitre se limite à la syntaxe de TVL, sa sémantique et l'analyse syntaxique et sémantique d'un fichier TVL. Il n'aborde pas les conséquences des cardinalités de feature sur des outils de faisabilité d'un modèle TVL tels qu'un Solver SAT ou SMP.

5.1 Motivations

Comme vu dans le chapitre 2 lors de la présentation des cardinalités de feature et du clonage, certains langages de la famille des feature diagrams permettent l'utilisation de cardinalités de feature (cfr 2.8).

Comme décrit par Czarnecki et al.[7], l'utilité des cardinalités de feature vient essentiellement des applications pratiques.

Parmi les applications pratiques, nous pouvons citer la modélisation des logiciels destinés aux systèmes embarqués, tels que les satellites, les téléphones portables, les appareils photos et bien d'autres outils électroniques. Ces systèmes embarqués sont très souvent déclinés en de nombreuses versions différentes, basées sur un socle commun et des composants réutilisables, il en va de même pour les logiciels qui y sont déployés. Ceux-ci représentent donc des exemples typiques de software product lines. Les feature diagrams peuvent donc être utilisés afin de représenter la variabilité de ces logiciels.

Voici un exemple issu de Czarnecki et al.[31], décrivant l'interface de communication des satellites avec les bases terrestres.

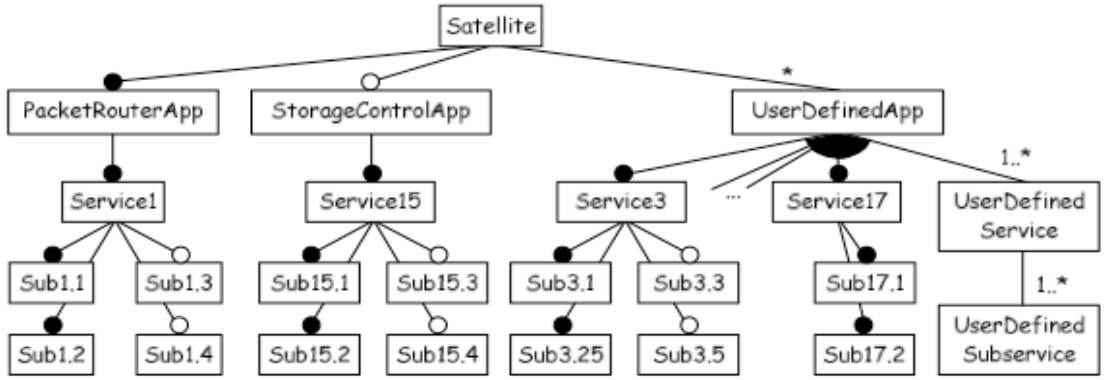


FIGURE 5.1 Extrait d'un feature diagram représentant un satellite [31]

Selon la représentation illustrée ci-dessus, tous les satellites disposent d'un système de routage ainsi que d'une collection d'applications offrant des services. Les applications déployées sur un satellite varient d'un satellite à l'autre, le nombre d'applications est non borné et chaque application peut aussi être composée d'un nombre non borné de services.

Les applications d'entreprises fournissent également des exemples d'applications nécessitant l'utilisation de cardinalités de feature.

Voici un exemple de système de gestion de la sécurité des fichiers dans une entreprise issu de Czarnecki et al. [7].

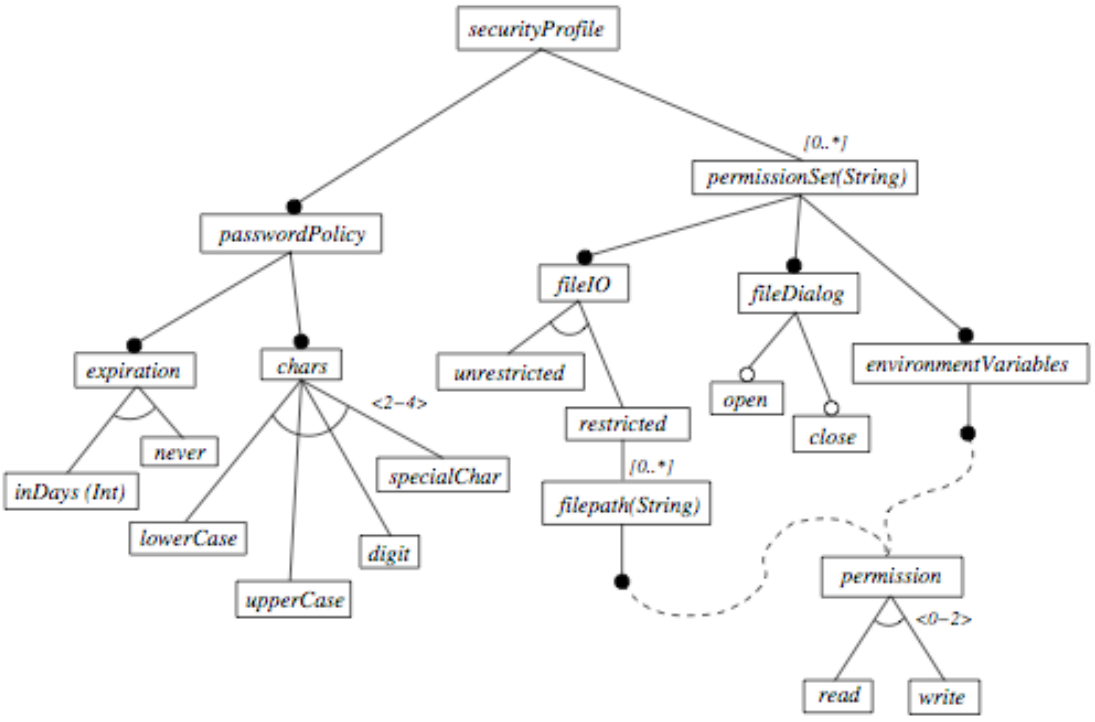


FIGURE 5.2 Extrait d'un feature diagram représentant un système de sécurité [7]

Pour rappel, un système classique de gestion des droits consiste à définir d'une part les permissions permettant d'effectuer des opérations sur des objets du système et d'autre part les utilisateurs du système. Les profils utilisateurs sont alors composés d'un ensemble de permissions et sont octroyés à un ensemble d'utilisateurs.

Le schéma 5.2 permet d'illustrer un profil utilisateur (SecurityProfile). Dans cet exemple, un profil est composé d'une collection d'ensembles de permissions (permissionSet) permettant d'effectuer des opérations (FileDialog) sur des fichiers (filePath). L'accès aux fichiers peut être illimité ou restreint à certains fichiers. Vu que le nombre de fichiers possibles est illimité, le feature « filepath » situé sous « restricted » est non borné.

La multiplicité définie sur un concept s'applique à ses descendants, chaque « permissionSet » est composé d'un « fileIO », d'un fileDialog et d'un « environnementsVariables ».

De plus, un « fileIO » d'un « permissionSet » porte sur plusieurs fichiers (filePath), une cardinalité est donc nécessaire au niveau de filePath.

Cet exemple, tout comme le précédent, illustre bien l'utilité de cardinalités de feature afin de représenter les multiplicités.

Les multiplicités doivent respecter trois propriétés:

- Elles peuvent être non bornées (ex : nombre non borné de programmes déployés sur la satellite, de fichiers sur le système informatique).
- Les multiplicités associées à un feature portent sur le sous-arbre de ce feature (ex : chaque permissionSet est composé d'un fileIO qui lui est propre)
- Des multiplicités peuvent être définies à différents niveaux de la hiérarchie (ex : permissionSet est caractérisé par des multiplicités, mais c'est aussi le cas de filepath).

La représentation de ces cardinalités de feature par une notation proche de UML permet de respecter ces propriétés.

La version de TVL antérieure à ce mémoire, décrite au chapitre 4, n'intègre pas les cardinalités de feature, mais vu leur utilité décrite par le biais des exemples précédents et vu que ces exemples sont des SPLs typiques que TVL doit être en mesure de représenter, les cardinalités de feature vont être ajoutées à TVL.

Cette problématique a déjà fait l'objet d'études. En effet, Michel et al. [22] propose une définition sémantique des cardinalités de feature, décrite à la section suivante.

5.2 Sémantique des cardinalités de Features

Comme vu au point 4.1, la sémantique de TVL est décrite grâce à une fonction sémantique $\mathcal{M}_{TVL} : \mathcal{L}_{TVL} \rightarrow \mathcal{S}_{TVL,}$ où $\mathcal{L}_{TVL}$ est le domaine syntaxique et $\mathcal{S}_{TVL}$ le domaine sémantique. Cette fonction sémantique associe donc une sémantique aux constructions issues de la syntaxe abstraite de TVL.

L'article de référence de Michel et al. [22] à propos de la sémantique des cardinalités de feature décrit en détails les modifications apportées à cette syntaxe abstraite ainsi qu'à la fonction sémantique et la sémantique. Cette section va présenter ces modifications.

Tout d'abord, un rappel de la notion de clonage vu lors de la description des feature diagrams (cfr 2.8), mais exprimée dans les termes de TVL, puis une présentation plus détaillée de la sémantique du clonage et des cardinalités de feature en tenant compte des spécificités et des choix effectués dans la sémantique de TVL. Cette présentation sera suivie d'une description plus formelle de cette sémantique.

5.2.1 Présentation du clonage

Le « clonage » est le terme désignant la possibilité d'instancier plusieurs occurrences d'un même feature dans un produit.

Par analogie aux langages orientés objet comme Java ou C#, le modèle TVL (l'expression d'un feature model en TVL) peut être vu comme la classe, alors que le produit peut être vu comme une instance de cette classe.

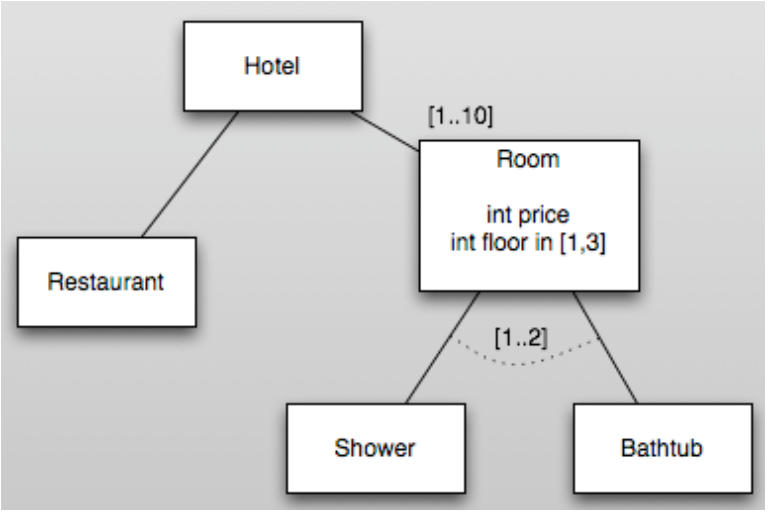
Dans un modèle TVL permettant le clonage, des features peuvent se voir attribuer des cardinalités de feature, celles-ci décrivent combien d'instances du feature peuvent être présentes dans un produit.

Lorsqu'un feature est instancié plusieurs fois dans un produit, ses instances sont appelées les clones de ce feature afin de rappeler le terme clonage.

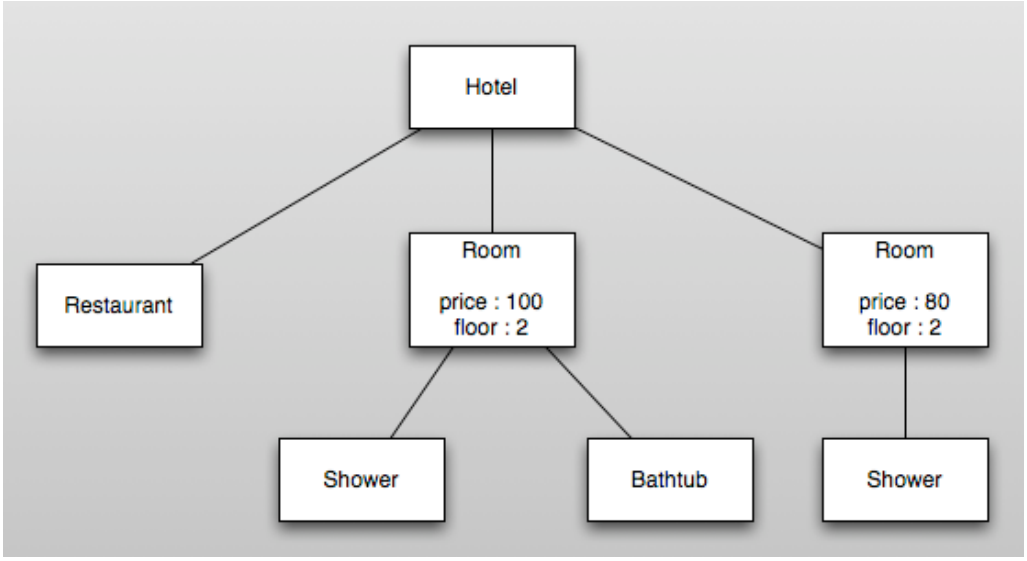
Toutefois, quand un feature du modèle est instancié une seule fois dans le produit, la définition de la sémantique utilise le terme feature pour désigner l'instance unique de ce feature.

Le produit est une instance du modèle TVL. Le terme produit est donc utilisé à plusieurs niveaux d'abstraction : les éléments d'une SPL sont appelés « les produits », les éléments d'une instance d'un modèle TVL sont aussi appelés les produits.

Afin d'illustrer le clonage de TVL, considérons une modèle représenté sous-forme d'un feature diagram (exemples issus des figures 2.12 et 2.13), il contient un feature « Room » clonable, ce feature contient des attributs et possède des sous-features.



Voici le schéma illustrant un produit TVL instanciant ce modèle :



Dans ce schéma, les nœuds « Room » sont des clones du feature « Room ». Ce schéma est valide, en effet :

- la cardinalité de feature [1..10] est respectée : 2 clones sont présents
- pour chaque clone, les types et valeurs des attributs sont conformes aux modèles, **bien que les valeurs soient différentes**
- pour chaque clone, la décomposition et la cardinalité de groupe sont respectées, **bien que différentes dans chaque clone.**

Comme introduit au point 2.8, l'utilisation de cardinalités de feature n'est pas une fonctionnalité triviale, elle a des répercussions sur différents aspects du langage, tels que les décompositions, les features optionnels ou encore les contraintes.

Le point 2.8 présentaient quelques questions relatives à l'introduction du clonage, la suite de ce chapitre doit y répondre, dans le cadre des choix effectués et détaillés par la sémantique de TVL.

L'impact du clonage sur les contraintes fera l'objet d'un chapitre à part entière (chap. 6).

Remarque :

Avant de décrire les conséquences du clonage sur les autres concepts de TVL, voici une remarque concernant le terme « clonage ».

Un clone au sens de TVL diffère d'un clone en Java. En effet, cloner un objet en Java génère un nouvel objet dont les valeurs des propriétés « primitives » sont identiques à l'objet initial, les propriétés de type « référence » référencent les mêmes objets (cfr Liskov et al., [27]).
Par contre, en TVL, différents clones d'un même feature peuvent avoir différentes valeurs et un nombre différent d'enfants.

La similitude réside dans le fait que les clones d'un même feature doivent respecter les mêmes règles imposées par le feature, tel que les noms des attributs, leurs contraintes sur les domaines de valeurs, les types de décomposition.

5.2.2 Clonage et cardinalités de groupe

Le point 4.2.2 décrit le fonctionnement des cardinalités de groupe sans clonage. Pour rappel, si les cardinalités de groupe d'un feature était [i..j], un produit contenant ce feature devait sélectionner entre i et j sous-features de ce feature.

Mais l'introduction du clonage change la donne. Comment calculer le nombre d'enfants du feature caractérisé par les cardinalités de groupe [i..j] ?

En effet, deux questions se posent :

- **Quel est le « scope » des cardinalités de groupe** ?
    - **Scope « clone »** : Les instances (ou clones) enfants du feature doivent être comptabilisées.
    - **Scope « feature »** : Le nombre de sous-features représentés par les instances (ou clones) présentes est comptabilisé.

  Exemple, si un parent P contient deux clones de types A, de combien incrémenter le compteur ?

  Si l'on compte le nombre de feature, le scope est dit « feature», le compteur sera incrémenter de 1 (car 1 type de feature : A)

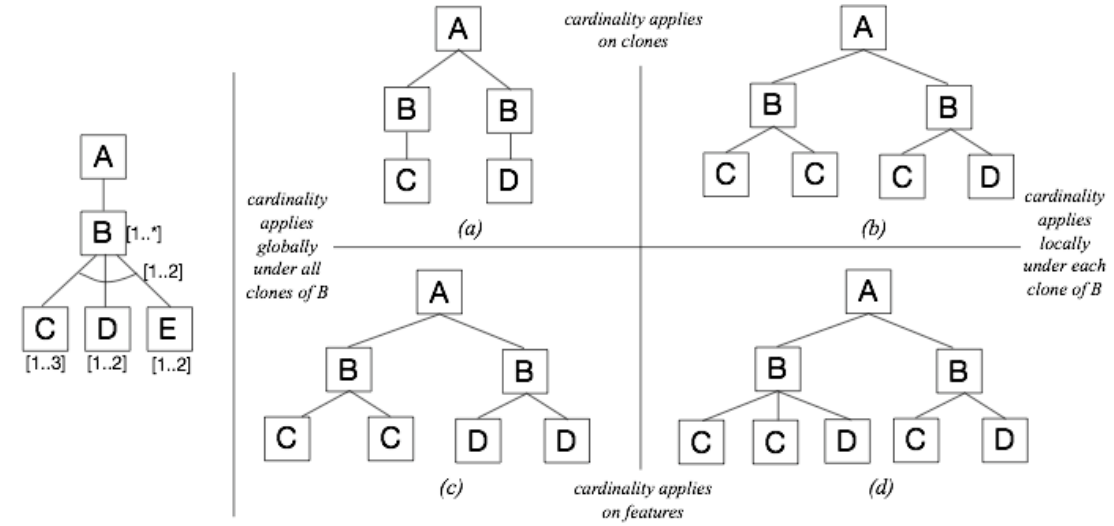
  Si l'on compte le nombre de clones, le scope est « clones», le compteur sera incrémenté de 2 (car 2 instances de A).

- **Quel est le « level » des cardinalités de groupe ?**
  Si le feature F, concerné par les cardinalités de groupe [i..j] possède plusieurs clones, à quel niveau vérifier les cardinalités ?
    - **Level « Local »** : Pour chaque clone du feature F, vérifier localement que le clone vérifie les cardinalités [i..j]
    - **Level «Global »** : compter globalement, sans tenir compte du parent.

  Par exemple, considérons un feature F de cardinalités de groupe [i..j] possédant deux clones : f1 et f2.

  Le scope local signifie que [i..j] est vérifié localement pour chaque clone. Le nombre de (clones ou features selon le scope) compté sous f1 doit respecter [i..j], le nombre compté sous f2 doit également respecter [i..j], mais ces deux nombres ne sont pas cumulés. Par contre, si le scope est global, les nombres comptés sous chaque clone sont cumulés. Et ce nombre total doit respecter les cardinalités.

Les choix effectués dans la sémantique de TVL sont le scope « feature » et le level « local ».



Cette figure 5.3 présente à gauche un modèle contenant un feature « B » dont les cardinalités de décomposition sont [1..2] et 3 sous-features. Sachant que la cardinalité maximale de décomposition de « B » est 2, elle illustre 4 cas atteignant ce maximum de 2, selon les 4 choix possibles à propos du scope et du level décrit ci dessus. Bien entendu, les cardinalités de feature présentes sur le schéma sont également respectées, mais le but de ces exemples est d'illustrer les choix possibles en termes de scope et level pour les cardinalités de groupe.

Les 4 exemples s'intéressent à la cardinalité maximale de décomposition de « B », qui est de 2.

Voici une brève explication de ces 4 cas :

- **(a) : Scope « Clone », level « Global »**
  Les instances de features (ou clones) sont comptées globalement : 2 clones : « C » et « D »

- **(b) : Scope « Clone », level «Local »**
  Les clones sont comptés localement sous chaque parent (« B ») : sous le 1er 2 clones C, sous le 2e 1 clone « C » et 1 clone « D », donc sous chaque parent, le maximum de 2 clones est atteint.

- **(c) : Scope « Feature », level « Global »**
  Les features sont comptabilisés, pas les clones. Et ce de manière globale. Donc 2 features sont représentés, « C » et « D », le maximum de 2 est atteint.

- **(d) : Scope « Feature », level « Local »**
  Les features sont comptés localement sous chaque parent (B) : sous le 1er, 2 features sont représentés (« C » et « D ») et sous le 2e, également 2 features représentés (« C » et « D »).

L'exemple (d) correspond aux choix de TVL. Cela correspond à la possibilité la plus souple.

En effet, le level « local » permet de traiter les clones indépendamment des clones sous d'autres parents. Un level « global » réduit le nombre d'enfants possibles sous chaque clone du parent, puisque les enfants de chaque clone parent sont cumulés.

De plus, ce level « local » correspond le mieux à la notion intuitive de clonage. L'utilisateur pourrait s'attendre à pouvoir cloner un feature et ses descendants tout en respectant les contraintes. Or, si le scope était « global », cloner un sous-arbre de features aurait une influence sur les clones déjà présents, puisque des nouveaux clones apparaîtraient et donc le nombre total de clones serait modifié.

Le scope « feature » offre également plus de liberté, puisque l'ajout d'un clone d'un sous feature n'a pas d'influence si le parent contient déjà un clone du même sous-feature.

Ce choix de scope « feature » est cohérent avec l'interaction des features optionnels et des cardinalités de groupe (cfr 4.2.2).

Pour rappel, considérons un modèle dans lequel un feature « F » dont la cardinalité minimale de décomposition est de 1 et composé, entre autres, d'un sous-feature optionnel « S ».

Un produit ne contenant aucun enfant sous « F » respecte la cardinalité de groupe, puisqu'un de ses sous-features est optionnel. De façon imagée, considérons que « F » sélectionne « S » dans sa décomposition, ce qui respecte le valeur minimale de 1, puis que « S » se « désélectionne », il le peut puisqu'il est optionnel et que ce caractère optionnel est prioritaire.

Nous pouvons imaginer un cas similaire où le sous-feature « S » est clonable. Le sélectionner dans la décomposition aurait une influence par rapport à la cardinalité de groupe, mais ajouter ensuite un clone du même type n'en aurait pas, tout comme supprimer une instance d'un feature optionnel n'en avait pas non plus dans le cas précédent.

5.2.3 Choix relatif aux cardinalités de Feature

Le point précédent décrit la façon de comptabiliser les enfants d'une instance (ou clone) d'un feature afin de vérifier ses cardinalités de groupe.

De façon similaire, la question du scope se pose également pour les cardinalités de feature.

En effet, les cardinalités de feature représentent le nombre de fois que le feature peut être instancié (« cloné »).

Si le produit possède une seule instance d'un feature « B », vérifier les cardinalités du feature « C » (enfant de « B ») est trivial, il suffit de compter les clones.

Par contre, si plusieurs instances du feature existent, comment comptabiliser les clones de C ? Deux solutions comparables au scope des cardinalités de groupe existent :

- **Scope global** : Vérifier les cardinalités localement pour chaque clone du parent B
- **Scope local** : Compter les clones de « C » globalement, indépendamment de leur parent.

Le schéma suivant illustre ces deux cas par un exemple de modèle, ainsi que par deux produits ayant atteint la cardinalité maximale, l'un selon le scope global et l'autre selon le scope local.

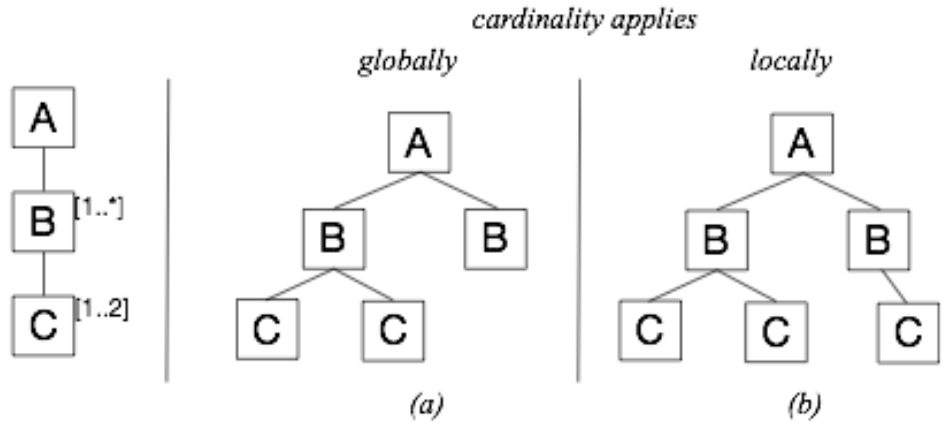


FIGURE 5.4 Produits possibles selon le scope des cardinalités de Feature (Michel et al. [22], p. 5)

Dans l'exemple (a), il existe globalement 2 clones de « C », le maximum de 2 défini dans la cardinalité de feature de « C » est donc atteint. Par contre, dans l'exemple (b), les clones sont comptabilisés localement, 2 clones de « C » sous le 1er parent « B » et 1 clone de « C » sous le 2e parent « B », donc la cardinalité maximale 2 est respectée sous chaque parent « B ».

La sémantique de TVL définit les cardinalités de feature avec un **scope local**.

Ce choix est cohérent avec celui effectué pour les cardinalités de groupe, il est tout comme lui plus proche de la notion intuitive de clonage et permet plus de liberté.

D'autres auteurs, comme Czarnecki et al. [11], ont également retenu cette solution.

5.2.4 Clonage et optionnalité

Le chapitre 4 (cfr 4.2.2) présente deux notions influençant le fait qu'un feature puisse être omis ou non. Avant d'expliquer les conséquences de l'introduction du clonage sur ces deux concepts, en voici un bref rappel.

Une décomposition d'un feature f est caractérisée par sa borne minimale i et sa borne maximale j, ce qui implique que dans un produit, le nombre n de sous-features de f sélectionnés doit être tel que i <= n <= j.

L'autre notion vue précédemment est celle de feature optionnel, un feature déclaré avec le mot-clé « opt » ne doit pas être obligatoirement sélectionné dans les produits, quelque soit les cardinalités de décomposition de son parent.

Ces deux notions peuvent être contradictoires, par exemple :

- Soient les cardinalités de décomposition [1..j] pour le feature f, avec 2 sous-features qui ne sont pas déclarés avec « opt ». Du point de vue de la décomposition, la présence d'un seul sous-feature est suffisante. Mais du point de vue du caractère optionnel, deux sous-features sont obligatoires.

- Soit les cardinalités de décomposition [i..j] avec i = j, avec des sous-features déclarés avec « opt ». Du point de vue de la décomposition, tous les sous-features doivent être sélectionnés, mais du point de vue du caractère optionnel, certains features peuvent être omis.

Le choix de TVL a été de privilégier le caractère optionnel. Un feature déclaré avec « opt » peut donc être omis, quelque soit la décomposition de son parent.

Un feature obligatoire (ce qui signifie : non déclaré avec « opt») pourra être omis si la décomposition dans laquelle il est inclus autorise à ne pas sélectionner tous les features, par exemple une décomposition OR ([1..j]).

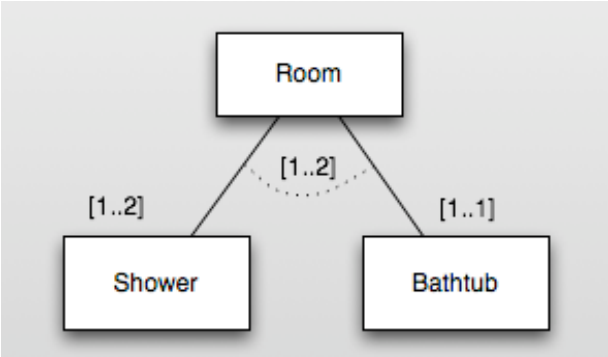
L'introduction de cardinalités de feature va induire une généralisation de ces cas.

Un feature obligatoire dans l'ancienne version de TVL est considéré comme un feature ayant une cardinalité [1..1] dans la nouvelle version, puisqu'il devait être présent et qu'il ne pouvait pas être cloné.

Par cohérence avec cette ancienne version, un feature déclaré avec la cardinalité de feature [1..1] pourra également être omis si la décomposition le permet ([i..j] avec i <j).

Cette règle est généralisée à tous les features obligatoires, c'est-à-dire les features dont la cardinalité de feature minimale est 1, quelque soit la cardinalité maximale.

Voici un petit exemple sous forme de feature diagram :



Dans ce schéma, les features « Shower » et « Bathtub » sont obligatoires (leur cardinalité minimale est supérieure à 0). Mais la décomposition est de [1..2], donc un produit ne comportant qu'un des deux sous-features parmi « Shower » et « Bathtub » serait valide.

Quant aux features dont la cardinalité minimale est 0, quelque soit leur cardinalité maximale, ils pourront être omis, peu importe la décomposition dans laquelle ils sont déclarés.

5.2.5 Tableau comparatif ancienne et nouvelle versions.

Avant d'examiner les modifications opérées à la définition formelle de la sémantique, voici une brève présentation des différences entre la version de TVL sans clonage et la version avec le clonage, citées dans les points précédents.

	TVL sans clonage	**TVL avec clonage**
Feature	Un feature peut être sélectionné ou non dans un produit.	Un feature peut être sélectionné plusieurs fois dans un produit, il convient donc de distinguer les features et les instances de features (clones).
Cardinalité de groupe du feature F	Indique combien de sous-features du feature F doivent être sélectionnés dans un produit.	Définition adaptée afin de tenir compte des features/instances : les cardinalités doivent être vérifiées sous chaque instance du feature F, en y comptant le nombre de features présents.
Cardinalité de feature de F	Pas défini, équivaut aux cardinalités de feature de [1..1] dans la version avec clonage.	Indique le nombre d'instances d'un feature F pouvant être présentes sous chaque instance du feature parent de F dans un produit.
Feature optionnel	Opt indique un feature pouvant être omis, équivaut dans cette version au terme feature optionnel.	Opt indique un feature de cardinalités [0..1]. **Attention**, le terme feature optionnel désigne un feature pouvant être omis, ce qui signifie dans cette version que la cardinalité minimale est 0, peu importe la cardinalité maximale.

5.2.6 Définition formelle

La sémantique de TVL (cfr articles de références de TVL [19,20] est adaptée afin d'inclure la définition des cardinalités de Feature).

Pour rappel, un modèle $d \in \mathcal{L}_{TVL}$ (syntaxe abstraite de TVL) est défini comme le tuple $(F, r, DE, \lambda, \omega, A, \rho, \tau, \Phi)$

- F : ensemble de features
- $r \in F$: racine
- $DE \subseteq F \times F$: la relation de décomposition entre les features.
  Par exemple, dans le couple $(p,e) \in DE$, noté aussi $p \rightarrow e$, p est le feature parent et e l'enfant de p.
- $\lambda : F \rightarrow N \times N$: fonction indiquant les cardinalités de décomposition d'un feature.
  Par exemple, $\lambda(f) = [m..n]$ signifie que la cardinalité minimale de décomposition du feature f est m, et la maximale est n.
- $\omega : F \rightarrow \{0, 1\}$: fonction indiquant si un feature est optionnel ou non.
  Par exemple, $\omega(f) = 0$ signifie que le feature f est optionnel
- A est l'ensemble des attributs
- $\rho : A \rightarrow F$ est une fonction qui retourne le feature auquel est attaché un attribut
- $\tau : A \rightarrow \{int, real, bool, enum, string\}$ est une fonction qui retourne le type d'un attribut
- Φ représente les contraintes « cross-tree »(cfr 4.2.4)

La nouvelle version de cette sémantique est expliquée en détail dans le document de référence de Michel et al. [22]. Cette section vise à présenter les modifications impactant la suite de ce mémoire.

- Le domaine syntaxique est adapté, la fonction $\omega : N \rightarrow \{0, 1\}$ indiquant si un feature est optionnel ou pas devient $\omega : N \times (N \cup \{*\})$, afin d'indiquer les cardinalités d'un feature.

- Dans la version précédente, le feature racine était obligatoire et le feature model ne pouvait comporter qu'une seule racine. Cette règle est adaptée aux cardinalités, le feature racine « r » est unique et ses cardinalités sont : $\omega(r) = [1..1]$.

- Un clone est défini comme ceci : "*If we define clone as a tuple (feature, children) where children is a multiset of clones, then the set of all possible clones is C such that* $C \subseteq$ *powerbag(C)*" (Michel et al. [22]) Cette définition implique qu'un clone ne possède qu'un seul parent (cfr 5.3.5 pour discussion dans le cadre de TVL). Attention, cela s'applique aux clones (les instances multiples de features), pas aux instances uniques (cfr 5.2.1 pour explications des termes clones et features).

La fonction sémantique est adaptée, elle se définit comme ceci (cfr Michel et al. [22]):

$\mathcal{M}_{TVL} : \mathcal{L}_{TVL} \rightarrow \mathcal{S}_{TVL}$ ($\mathcal{L}_{TVL}$ et $\mathcal{S}_{TVL}$ sont les domaines syntaxique et sémantique, cfr 4.1)

où $\mathcal{M}_{TVL}$(d) est le set de tous les produits issus du modèle d tels que $\forall p \in \mathcal{M}_{TVL}$(d) :

- p = (r, D)
  Ceci est la représentation d'un clone p, instance du feature r (racine du modèle) et dont les enfants sont dans le multiset D.
- p |= d
  p respecte les contraintes (au sens large, pas uniquement les cross-tree) du modèle

La définition des règles à vérifier pour qu'un clone quelconque, instance d'un feature f et dont les enfants sont notés D, est adaptée :

(f, D) |= d ⇔

- Nouvelle règle de respect de la décomposition hiérarchique

  $$\forall (g, E) \in D : f \rightarrow g$$

  Les instances d'un feature (ou « clones ») présents dans le produit doivent respecter la même décomposition que le feature correspondant dans le modèle.

- Adaptation de la règle de définition des cardinalités de décomposition : Soit [m..n] les cardinalités de décomposition d'un feature f :

  $$m - |\ opt_N\ | \leq |\ mand_p\ | \text{ et } |\ all_p\ | \leq n$$

  Avec :
  - $opt_F = \{g \mid g \in F \wedge \mathbf{\omega(g) = [0..y]} \wedge f \rightarrow g\}$
  - $all_p = \{g \mid \exists (g, E) \in D\}$
  - $mand_p = all_p \setminus opt_F$

Remarques :

- **ω(g) = [0..y]** indique que les features optionnels sont rechercher d'après la valeur de la cardinalité minimale.

- Ce sont bien les features que l'on comptabilise, et ce localement.

    - $g \in F$ indique que les éléments comptabilisés appartiennent à l'ensemble des features.
    - $\exists (g, E) \in D$ est vérifié s'il existe un clone du feature g dans le produit, mais c'est bien le nombre de features qui est pris en compte.

- Ajout d'une règle de définition des cardinalités de feature :

  $\forall g : f \rightarrow g : \omega(g) = \langle m..n \rangle$
  $\Rightarrow clones_g = 0 \vee m \leq clones_g \leq n$

  avec: clones = $|\{|(g, E) \in D|\}|$ signifiant le nombre d'instances du feature g

  Remarques :
  - $clones_g = 0$ est vrai si aucune instance du feature n'est présente. Donc la valeur des cardinalités n'ont pas d'importance si le feature n'a pas été instancié, même si la cardinalité minimale est supérieure à 0 (cela correspond au point discuté en 5.2.4).
  - les cardinalités de feature se vérifient localement

- Cette définition est récursive, les conditions énuérées ci-dessus doivent donc être vérifiées pour chaque enfants de f : $\forall (g, E) \in D : (g, E) \models d$

En outre, p devrait satisfaire Φ (« cross-tree constraints »), mais la syntaxe et la sémantique de ces contraintes doivent être revues suite à l'introduction du clonage. Une proposition sera faite dans ce sens au chapitre 6.

5.2.7 Enumération des exigences de la sémantique

Après avoir revu les spécifications du langage abstrait et de sa sémantique (cfr 5.2), voici une énumération des points à considérer comme les exigences en vue de l'élaboration des spécifications des modifications à apporter à la syntaxe concrète de TVL ainsi qu'à son module d'analyse sémantique.

- **Cardinalités de features**
  - Les cardinalités sont définies sous la forme [i..j] ou [i..*], avec i et j entiers positifs, tels que i <= j
  - La vérification de ces cardinalités se fait localement sous chaque parent.
  - Si le feature ne possède aucun clone sous un parent, les cardinalités seront considérées comme vérifiées sous ce parent, même si la cardinalité minimale est supérieure à 0.

- **Cardinalités de décomposition**
  - La vérification des cardinalités de décomposition se fait localement sous chaque parent et seul le nombre de features représentés importe, pas le nombre de clones.

- **Racine :** Elle est obligatoire et ne peut pas être clonée (cardinalité [1..1]).

- **Optionalité :** Un feature est optionnel si sa cardinalité de feature minimale est 0, quelque soit la valeur maximale (cardinalité [0..y]).

- **Clone** : Chaque clone doit respecter la hiérarchie définie par le modèle. De plus, un clone ne pourra posséder qu'un seul parent.

5.3 Transpositions des exigences à TVL

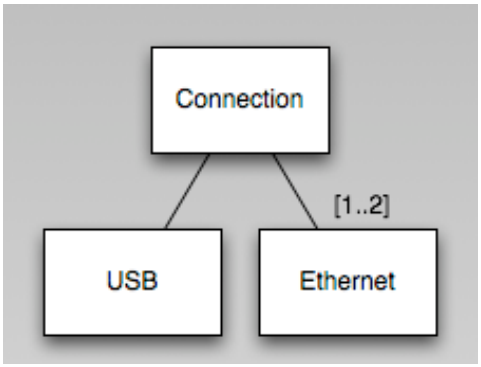
Cette section vise à traduire les exigences issues de la sémantique et du langage abstrait en spécifications de la syntaxe concrète de TVL et de règles de corrections sémantiques.

Les modifications apportées à TVL sont présentées par des exemples de code TVL.

Les adaptations de la grammaire formelle de TVL seront présentées dans la section suivante (cfr 5.4).

5.3.1 Expression des cardinalités de Feature

Ces cardinalités seront exprimées sous la forme [i..j] à la suite du nom du feature lors de sa déclaration.



Par exemple, reprenons le cas de la figure 2.11

Le modèle TVL est le suivant :

```
root Connection group allOf{
      USB,
      Ethernet [1..2]
}
```

Plusieurs règles sont à vérifier à propos de ces cardinalités [i..j] :

- i est un entier positif
- j est un entier positif ou le symbole *
- i <= j, l'opérateur <= étant étendu de sorte que i <= * soit vrai.

De plus, la déclaration de [i..j] ne peut se faire que lors de la première déclaration du feature sous son parent.

En effet, TVL autorise l'extension de la définition du corps d'un feature (cfr 4.2.1).

Par exemple, ajoutons un attribut à « Ethernet » en utilisant le principe d'extension :

```
root Connection group allOf{
      USB,
      Ethernet [1..2]
}
Ethernet {
      int value ;
}
```

La définition des cardinalités de feature ne peut donc se faire que lors de la déclaration de « Ethernet » dans la hiérarchie de son parent, ce qui semble intuitif.

Mais cette décision ne repose pas uniquement sur l'intuition, elle repose sur deux choix :

- Une seule déclaration des cardinalités est autorisée.
- De plus, cette déclaration doit se faire lors de la déclaration du feature sous son parent.

Autoriser plusieurs déclarations de cardinalités serait source d'erreurs, par exemple :

```
root Connection group allOf{
     USB,
     Ethernet
}

Ethernet [1..2] { int value ; }
Ethernet { int color ; }
Ethernet [1..3] { int mark ; }
```

Quel serait dans ce cas les cardinalités à retenir, [1..1], [1..2] ou [1..3] ?

La présence de plusieurs valeurs de cardinalités de features pose un problème de lisibilité, elle est à elle seule une raison suffisante pour le rejet d'une déclaration multiple de cardinalités.

Autre raison de refus, une notion d'ordre dans les déclarations permettrait par exemple de sélectionner les premières cardinalités rencontrées. Mais cette notion d'ordre n'existe pas actuellement dans TVL ni sans sa syntaxe abstraite.

Autoriser une déclaration par extension poserait également problème :

Par cohérence et compatibilité avec l'ancienne version de TVL, un feature sans cardinalité de feature reçoit par défaut les cardinalités [1..1].

Dans l'exemple ci-dessus, la première déclaration de « Ethernet » sous son parent, sans cardinalité de feature, correspond donc à une déclaration [1..1]. Toute déclaration en extension serait donc une déclaration supplémentaire et serait donc refusée vu l'interdiction de fournir plusieurs déclarations de cardinalités de feature.

Une autre solution pourrait-elle être de ne n'octroyer la cardinalité par défaut à un feature que si : ni sa déclaration sous son parent, ni aucune de ses extensions ne spécifient explicitement de cardinalités ? Dans ce cas, l'absence de cardinalité ne signifierait pas toujours l'application des cardinalités par défaut. L'utilisateur devrait donc lire tout le modèle TVL afin de trouver la définition et toutes les extensions d'un feature, il pourrait alors seulement en déduire les cardinalités, ceci n'est pas acceptable.

De plus, déclarer les cardinalités associées à un feature lors de sa déclaration sous son parent est cohérent avec l'utilisation du mot clé « opt » permettant de déclarer les features optionnels.

Enfin, déclarer les cardinalités de feature lors de n'importe quelle extension et/ou les déclarer plusieurs fois n'apporterait rien sur le plan sémantique.

5.3.2 Cardinalités de groupe

La syntaxe des cardinalités de groupe (« ou de décomposition ») n'est pas modifiée.

Toutefois, leur sémantique a été modifiée puisque l'ancienne version de la sémantique ne différenciait pas les notions de clones et de features. Dans la nouvelle sémantique, les cardinalités de groupe sont vérifiées localement et en comptant les features représentés et non les clones (cfr 5.2.2).

Cette vérification n'influence pas la modification du module d'analyse sémantique de TVL, puisque ce module vise à vérifier la correction d'un modèle.

Le dénombrement d'instances de feature se fait dans le cas de la vérification de la validité d'un produit par rapport à un modèle, ceci est abordé dans les chapitres 7 et 8, relatifs à la description d'un langage de configuration de modèle et d'un outil de validation d'un produit par rapport au modèle dont il est dérivé. Dans ce cas, nous verrons que la description sémantique des cardinalités de groupe est importante.

5.3.3 Feature racine (« root »)

Pas de modification syntaxique, le feature racine est précédé de « root » et ne peut pas être défini en extension.

L'utilisation des cardinalités de feature sous la forme [x..y] pour ce feature racine n'est pas permise, ses cardinalités au niveau sémantique seront considérées comme étant celles par défaut, donc [1..1].

Pourquoi cette limitation ?

Les exigences issues de la sémantique de TVL imposent que le feature racine soit de cardinalités [1..1].

Donc si la syntaxe autorisait l'utilisation de cardinalités [x..y], x et y devraient obligatoirement être de valeur 1.

Sémantiquement, la présence ou pas de [1..1] dans la syntaxe ne change rien.

Ergonomiquement, le [1..1] pourrait rappeler à l'utilisateur que la racine est de cardinalités [1..1], mais cela pourrait aussi l'induire en erreur, il pourrait oublier que ces valeurs ne peuvent pas être modifiées contrairement aux autres features ayant des cardinalités définies explicitement.

Vu ces éléments sémantique et ergonomique et afin de simplifier la syntaxe et les traitements d'analyse sémantique, la présence de cardinalités explicites pour le feature racine n'est pas autorisée.

5.3.4 Feature optionnel (« opt »)

Suite à l'introduction de cardinalités de feature permettant le clonage, voici quelques précisions à propos de ce que l'on entend par feature optionnel et à propos du mot clé « opt » permettant de marquer les features optionnels dans l'ancienne version de TVL.

Dans la version précédente de TVL, un feature était soit optionnel, soit obligatoire, le clonage n'existait pas, on ne se posait donc pas de question à propos du nombre maximum d'instance du feature, mais uniquement à propos du nombre minimum.

Vu l'introduction de cardinalités de feature, un feature peut être cloné. Lorsque l'on parlera à l'avenir de feature optionnel, cela signifiera un feature **dont la cardinalité minimale est 0, quelque soit sa cardinalité maximale**.

Dans l'ancienne version, le mot clé « opt » signifiait que le feature était « optionnel ». Mais dans ce cas, le sens de « optionnel » prête à confusion.

En effet, lorsque l'on utilise le terme « optionnel » dans la nouvelle sémantique de TVL, cela signifie que la cardinalité de feature minimale est 0.

Que faire du mot clé « opt » ?

Par compatibilité avec l'ancienne version, ce mot clé « opt» est permis dans la nouvelle syntaxe.

Ce mot clé « opt » aura la même sémantique que des cardinalités de feature [0..1], puisqu'il revient à dire que le feature est optionnel (dans le sens général du terme, minimum est 0) et que la cardinalité maximale n'est pas déclarée (donc 1 par défaut).

Par contre, un cumul ne pourra à la fois utilisé « opt » et une déclaration explicite de cardinalités sous la forme [x..y].

Quelle est la confusion ?

« opt » a le même sens dans la nouvelle version que dans l'ancienne, il correspond à une cardinalité [0..1], c'est donc cohérent.

Le risque de confusion vient de la généralisation de sens de mot optionnel. Un feature marqué comme « opt » est optionnel (sa cardinalité minimale est bien 0), par contre un feature optionnel n'est pas équivalent à un feature déclaré avec « opt », puisque le mot clé « opt » implique une cardinalité maximale de 1.

Ce risque de confusion devra être pris en compte lors des spécifications et de l'implémentation des analyseurs syntaxique et sémantique de TVL.

Par exemple, que faire d'une méthode spécifiée à l'époque de l'ancienne version de TVL dont la spécification serait de retourner « true » si le feature est « optionnel » ? Si cela ne prêtait pas à confusion dans l'ancienne version de TVL, ce n'est plus le cas. Doit-on se baser sur la cardinalité minimale ou se baser sur le fait que le feature soit marqué dans le code du modèle TVL avec le mot clé « opt ».

Les nouvelles spécifications devront éviter cette ambiguïté, et les spécifications existantes devront être adaptées selon le contexte en cas d'ambiguïté.

5.3.5 Feature partagé (« shared »)

Dans la sémantique abstraite des feature models adaptée suite à l'introduction du clonage, la définition d'un clone interdit qu'un clone possède plusieurs parents.

Pour rappel, le terme clone désigne une instance d'un feature dans le cas d'une instanciation multiple.

La sémantique de TVL se base sur cette sémantique abstraite, donc au niveau de la syntaxe concrète de TVL, un feature partagé ne pourra pas être cloné.

En effet, si un feature est partagé, il dispose de plusieurs parents. Or, un clone ne peut pas disposer de plusieurs parents. Donc, un feature partagé ne peut pas être cloné.

Par contre, si le feature n'est instancié qu'une seule fois, il peut être partagé.

En d'autres termes, un feature partagé n'est pas clonable, voici un exemple :

```
House group allOf{
      Room {Door},
      BathRoom {shared Door}
      Hall {shared Door}
}
```

Dans ce 1er exemple, le feature « Door » est partagé par ses parents : « Room », « BathRoom » et « Hall ». Il est déclaré sous un parent sans « shared » et sous tous les autres avec le mot-clé « shared ». Ce feature « Door » ne pourra pas utiliser de cardinalités de feature. D'un point de vue sémantique, il héritera des cardinalités par défaut [1..1].

Ceci peut sembler être un manque d'expressivité, mais en pratique, les cas industriels nécessitent très rarement l'utilisation de features partagés (cfr Michel et al. [22]).

Cette possibilité de partage de feature est donc jugée très peu utile. De plus, elle complexifie la syntaxe mais aussi les règles sémantiques de validation du modèle et des produits.

Voici un exemple afin d'illustrer la complexité engendrée par la présence de features partagés :

Reprenons l'exemple ci-dessus en ajoutant des cardinalités au feature « Door », par exemple [3..3], ce qui obligerait les produits à disposer de 3 clones de « Door ».

Un produit contiendrait donc 3 clones de « Door », chacun devant être relié à un clone de chaque parent afin de respecter la décomposition. Les features « Room », « BathRoom » et « Hall » devraient donc aussi être clonables.

Cela implique une compatibilité des cardinalités de feature des différents parents avec celle du feature partagé. Par exemple, si un des parents a une cardinalité maximale de 2, un produit ne pourrait pas contenir 3 clones de ce parent alors que 3 clones seraient nécessaires pour satisfaire les 3 clones du feature enfant.

L'analyse sémantique du modèle devrait donc vérifier cette compatibilité. De plus, la syntaxe devrait définir sur quelle déclaration de « Door » préciser les cardinalités.

Cette contrainte interdisant de cloner les features shared va être étendue par TVL, les parents d'un feature partagé ne pourront pas être clonés.

Imaginons un cas ou le feature shared n'est pas clonable mais où ses parents le seraient, par exemple :

```
House group allOf {
      Room [1..3] group allOf {Door},
      BathRoom [1..2] group allOf {shared Door}
}
```

Si un produit issu de ce modèle contient 2 clones de « Room » et 2 clones des « BathRoom », il devrait aussi contenir deux clones de « Door », chacun de ces clones de « Door » étant partagé entre à 1 clone de « Room » et 1 clone de « BathRoom ».

Mais puisque le feature « Door » n'est pas clonable, il n'y aurait pas assez de clones de « Door » pour tous les clones des parents, ce produit serait donc invalide.

De plus, si les cardinalités des parents sont différentes, ou si elles sont identiques mais que le produit ne contient pas le même nombre de chaque parent, le produit serait invalide, même sans tenir compte du nombre d'enfants.

En conclusion, vu la faible utilité des features partagés dans les cas pratiques et vu la complexité qu'ils engendrent dans différents cas, **la version de TVL permettant le clonage ne permettra pas de cloner un feature partagé, ni ses parents.**

En effet, l'avantage de TVL par rapport à d'autres langages de modélisation plus « généraliste » tels que UML et OCL est qu'il est relativement simple. Sa syntaxe comporte peu de termes, peu de règles et permet une prise en main rapide des utilisateurs souhaitant écrire des modèles TVL ainsi qu'une bonne compréhension intuitive de la part des non-initiés.

Complexifier TVL pour résoudre des petits cas possibles théoriquement mais sans réelle importance dans la pratique est donc à éviter, bien que ce soit une tendance naturelle chez les développeurs.

En effet, à force de complexifier sans cesse le langage pour y traiter des cas de plus en plus petits et diversifiés, le langage finirait par traiter des cas bien éloignés de la réelle utilité du langage, avec une complexité de plus en plus élevée, le rendant de plus en plus difficile à utiliser.

5.4 Adaptation de la Grammaire de TVL

Comme vu au chapitre 3, la grammaire d'un langage définit sa syntaxe concrète. Cette section vise à exprimer, sous forme de grammaire, les modifications de la syntaxe concrète de TVL décrites par la section précédente.

5.4.1 Présentation de la nouvelle version de la grammaire

Cette grammaire est ici présentée sous la forme EBNF (Extend Backus-Naur form). Cette forme étendue de BNF (la norme standard de représentation des grammaires) offre quelques « sucres syntaxiques » :

- Les ( ) sont utilisées pour grouper des éléments
- [S] signifie que l'élément S est optionnel
- (S) + signifie que S se répète 1 ou plusieurs fois
- (S) * signifie que S se répète 0 ou plusieurs fois

Pour clarifier la notation, les non-terminaux sont en majuscules et les terminaux en minuscules et entre « ».

Les modifications de cette grammaire portent sur les éléments permettant de définir les features, le but est d'y inclure la définition des cardinalités de feature tout en respectant au mieux les contraintes d'utilisation de ces cardinalités. Les contraintes non vérifiables par l'analyse syntaxique seront traitées par l'analyse sémantique. L'ancienne version de la grammaire est décrite dans « Syntax & Semantic of TVL » [20].

Pour rappel, un fichier TVL est constitué de déclarations de features et de leur corps. Un feature du feature model est donc déclaré sous la décomposition de son parent, mais peut en plus être défini en extension.

Le fichier TVL contient donc la déclaration d'un feature racine, sous lequel des sous-features peuvent être présents, pouvant eux mêmes être décomposés en sous-features et ainsi de suite.

Mais le fichier contient aussi des définitions de features en extension, ces features doivent aussi être déclarés sous la décomposition de leur parent et peuvent lors de leurs définitions en extension définir leur sous-features, si ce n'a pas été fait sous la déclaration du feature sous le parent ou dans une autre définition en extension.

Dans la grammaire, FEATURE représente une déclaration d'un feature racine ou d'une extension de la définition d'un feature. Si l'on considère le sous-ensemble de la grammaire destiné à déclarer les features, leurs attributs et contraintes, FEATURE est l'axiome de départ de ce sous-ensemble.

```
FEATURE ::= FEATURE_ROOT | FEATURE_NODE ;
```

Comme dans l'ancienne version, un feature est composé de son nom simple, (ID) si déclaré sous son parent, ou nom qualifié (LONG_ID) dans le cas de l'extension. Ensuite, soit son corps : (FEATURE_BODY) soit sa décomposition (FEATURE_GROUP).

Le corps est lui composé d'attributs (ATTRIBUTE), de contraintes (CONSTRAINT), de données (DATA) et de sa décomposition (FEATURE_GROUP).

Les cardinalités ne peuvent pas être utilisées pour le feature racine, ni pour les définitions en extension, voici donc les règles décrivant la déclaration du feature root et d'un feature sans cardinalité.

```
FEATURE_ROOT ::= «root» ID «{» FEATURE_BODY «}»| «root» ID FEATURE_GROUP ;
FEATURE_NODE ::= LONG_ID «{»  FEATURE_BODY «}» | LONG_ID FEATURE_GROUP ;
```

Voici maintenant le corps des features, contenant une liste de données, attributs, contraintes et décomposition :

```
FEATURE_BODY ::= (DATA|CONSTRAINT|ATTRIBUTE|FEATURE_GROUP) * ;
```

D'un point de vue syntaxique, le feature pourrait donc contenir plusieurs déclarations de sa décomposition, ce qui est pourtant interdit par TVL, cette interdiction sera donc gérée par l'analyse sémantique.

La décomposition est composée d'une liste de déclarations de features, ce sont les features enfants.

```
FEATURE_GROUP ::= «group» CARDINALITY_GROUP «{» (HIERARCHICAL_FEATURE)* «}»;
```

Une feature enfant est déclaré de la façon suivante :

```
HIERARCHICAL_FEATURE ::=  LONG_ID [CARDINALITY_FEATURE]          (1)
                        | (« opt » | « shared ») LONG_ID         (2)
                        | FEATURE_NODE                           (3)
                        | FEATURE_NODE_CLONE                     (4)
                        | (« opt » | « shared ») FEATURE_NODE ;  (5)
```

La définition d'un enfant peut se limiter à la déclaration de son nom et de ses cardinalités s'il en possède, d'où les règles (1) et (2). La déclaration peut aussi définir le corps du feature, voir (3), (4), (5).

La définition d'un feature enfant peut aussi déclarer le corps ou la décomposition d'un de ce feature de façon comparable à la déclaration des features déclarer au 1er niveau du fichier. La FEATURE_NODE est donc également permis (3).

Mais contrairement aux déclarations d'extensions, les cardinalités sont permises sous la décomposition du parent, le FEATURE_NODE_CLONE est donc permis, dont voici la définition.

```
FEATURE_NODE_CLONE ::=  LONG_ID CARDINALITY_FEATURE « { »  FEATURE_BODY « } »
                        LONG_ID CARDINALITY_FEATURE FEATURE_GROUP ;
```

Avec les règles suivantes pour les cardinalités :

```
CARDINALITY_FEATURE ::= «[» CARDINALITY_LIMIT «..» CARDINALITY_LIMIT «]» ;
CARDINALITY_LIMIT ::= INTEGER | «*» ;
```

CARDINALITY_FEATURE définit le format des cardinalités de feature, ce format est comparable à celui des cardinalités de groupe, mais l'utilisation de mots clés « someOf », « allOf » et « oneOf » n'est pas permise puisque réservée aux décompositions.

5.4.2 Conclusion

Les modifications de cette grammaire peuvent donc se résumer en la distinction de FEATURE_NODE et FEATURE_NODE_CLONE afin de permettre la définition avec clonage ou sans clonage sous une décomposition, mais en ne permettant d'utiliser que la version sans le clonage dans une définition en extension. La présence de « root » ou « opt » n'autorise pas l'utilisation de cardinalités, les contraintes concernant la racine (cfr 5.3.3) et les features optionnels (5.3.4) sont donc vérifiées.

Cette grammaire ne vérifie pas toutes les contraintes décrites dans la description de TVL (cfr chapitre 4), mais ces contraintes existaient déjà dans l'ancienne version de la syntaxe et dans l'analyseur sémantique de TVL, elles seront conservées dans la nouvelle version.

Par contre, l'introduction des cardinalités de feature entraîne l'apparition de nouvelles contraintes ne pouvant pas être vérifiées par la grammaire :

- validité des bornes de cardinalités de feature
- interdiction d'utiliser les cardinalités de feature avec un feature partagé (cfr 5.3.5)

La première est trivial et sera facilement implémentée dans l'analyseur syntaxique.

Concernant la seconde, la grammaire interdit l'utilisation de cardinalités lors de la déclaration d'un feature précédé par le mot « shared », mais ce n'est pas suffisant.

En effet, comme vu dans le chapitre 4 décrivant l'ancienne version de TVL (cfr 4.2.6), le feature partagé est déclaré sous chacun de ses parents. S'il possède N parents, il est déclaré sous N-1 de ses parents avec le mot clé « shared », mais sous 1 parent sans ce mot clé. La grammaire autorise donc à préciser une cardinalité sous ce parent. L'analyseur syntaxique devra donc signaler une erreur s'il découvre une cardinalité sous ce parent.

Voici une vue plus compacte de la portion de la grammaire concernant les déclarations de features :

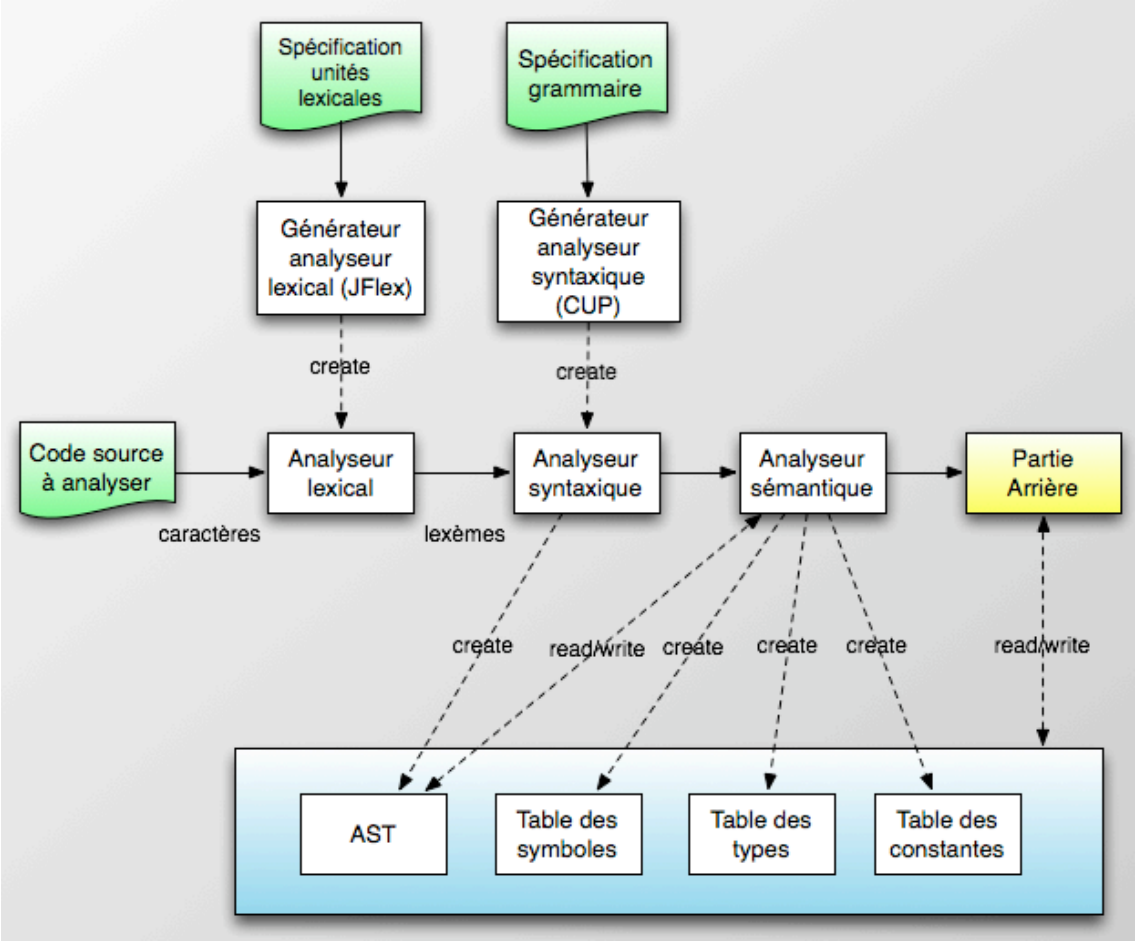
```
FEATURE ::= FEATURE_ROOT | FEATURE_NODE ;
FEATURE_ROOT ::= «root»  ID «{» FEATURE_BODY « } » | root ID FEATURE_GROUP ;
FEATURE_NODE ::= LONG_ID «{»  FEATURE_BODY « } » | LONG_ID FEATURE_GROUP ;

FEATURE_BODY ::= (DATA|CONSTRAINT|ATTRIBUTE|FEATURE_GROUP) * ;
FEATURE_GROUP ::= «group» CARDINALITY_GROUP «{» (HIERARCHICAL_FEATURE)* «}» ;

FEATURE_NODE_CLONE ::=  LONG_ID CARDINALITY_FEATURE «{»  FEATURE_BODY «}»
                                | LONG_ID CARDINALITY_FEATURE FEATURE_GROUP ;

HIERARCHICAL_FEATURE ::=   LONG_ID [CARDINALITY_FEATURE]
                         | («opt» | «shared») LONG_ID
                         | FEATURE_NODE
                         | FEATURE_NODE_CLONE
                         | («opt» | «shared») FEATURE_NODE ;

CARDINALITY_FEATURE ::=  «[»  CARDINALITY_LIMIT «..» CARDINALITY_LIMIT «]» ;
CARDINALITY_LIMIT ::= INTEGER | «*» ;
```

Les déclarations des attributs (ATTRIBUT) et des datas (DATA) restent inchangées, tout comme la déclaration des cardinalités de groupe (CARDINALITY_FEATURE) (cfr articles de références de TVL [19,20]).

Par contre, la portion de la grammaire définissant la syntaxe des expressions de définitions des contraintes sera revue au chapitre 6.

5.5 Implémentation

Cette section présente les modifications apportées à l'implémentation de TVL. Afin de les expliquer, reprenons la figure 4.3 présentant la structure du parser TVL.



L'analyseur lexical est généré par un générateur se basant sur la spécification des unités lexicales. Aucun nouveau mot clé n'a été ajouté, il n'y a donc pas de modification au niveau lexicale.

Au niveau syntaxique, la grammaire a été modifiée (cfr 5.4), elle permettra de générer la nouvelle version de l'analyseur syntaxique.

La représentation interne doit être adaptée afin de permettre à l'analyseur syntaxique de stocker les cardinalités de feature dans l'arbre syntaxique (AST) et dans la table des symboles.

L'analyseur sémantique, responsable de la construction de la table des symboles et de la vérification des contraintes sémantiques (cfr 3.5.2) doit aussi être adapté.

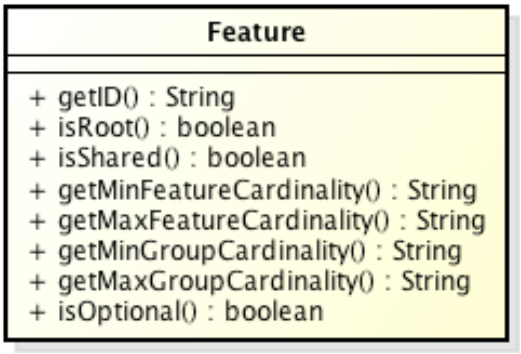
Les points suivants présenteront donc les modifications de l'AST, puis la modification du processus de construction de la table des symboles.

La structure originale de l'AST et celle de la table de symboles sont décrites dans le mémoire de P. Faber [24].

5.5.1 Structure de l'AST

Chaque définition d'un feature, que ce soit sous son parent ou en extension, est stockée dans un objet de type Feature dans l'AST.

La classe Feature, du package be.ac.fundp.info.TVLParser.SyntaxTree, est adaptée et implémentera les méthodes suivantes :



(Rem : Par soucis de lisibilité, les constructeurs ne sont pas représentés sur ce schéma)

Les deux nouvelles méthodes getMinFeatureCardinality et getMaxFeatureCardinality retournent les valeurs minimales et maximales des cardinalités de feature. Si les cardinalités ne sont pas précisées dans le fichier TVL de manière explicite, ces méthodes retournent une string vide.

La méthode isOptional indique si le feature est déclaré avec le mot clé « opt ».

La classe contient 3 familles de constructeurs utilisées par l'analyseur syntaxique:

- Une première recevant 4 paramètres :
  - ID du feature
  - true s'il s'agit du feature root, false sinon
  - true si le mot clé shared est présent, false sinon
  - true si le mot clé « opt » est présent, false sinon
- Idem que la première, mais avec un objet de classe FeatureBody.
- Idem que la première, mais avec un objet de type FeatureGroup.

FeatureBody et FeatureGroup sont également des classes de l'AST. Elles contiennent respectivement les informations relatives au corps d'un feature et à la décomposition d'un feature. Ces deux classes sont produites par l'analyseur syntaxique lorsqu'il rencontre respectivement une déclaration de corps de feature et une déclaration d'une décomposition de feature (cfr 5.4.1 description de la grammaire).

Pour chaque famille, il existe deux constructeurs, l'un avec et l'autre sans cardinalité.

5.5.2 Construction de la table des symboles

Lors du processus de construction de la table des symboles, TVL parcourt l'AST afin de créer, dans la table des symboles, un nœud par feature du modèle, ce nœud est de type FeatureSymbol (dans la package be.ac.fundp.info.TVLParser.symbolTables).

Ces nœuds contiennent toutes les informations à propos des attributs, contraintes et datas définies dans l'AST.

Dans l'AST, un feature peut être décrit par plusieurs nœuds différents, puisqu'il peut être défini en extension. Mais dans la table des symboles, toutes les informations issues de différentes définitions d'un feature sont regroupées dans le nœud FeatureSymbol associé à ce feature.

Les modifications suivantes sont apportées à cette 1ere phase de construction :

Pour chaque nœud de l'AST représentant une déclaration d'un feature sous son feature parent, les cardinalités du FeatureSymbol associé au Feature vont être garnies, deux types de cardinalités sont à distinguer :

- **Cardinalités explicites**

  Les cardinalités définies explicitement dans le fichier TVL sous la forme [i..j] et donc reprises dans l'AST sont vérifiées (pour [i..j], vérifier que i<=j ou j = *) et si elles sont valides, elles sont stockées dans le FeatureSymbol associé au feature auquel elle sont associées dans l'AST. En cas de valeurs invalides, une exception est lancée.

- **Cardinalités implicites**

  Comme vu précédemment, les cardinalités de feature ne peuvent pas être utilisées pour des features déclarés avec les mots-clés « root » (cfr 5.2.3) ou « opt » (cfr 5.2.4). Sémantiquement, le feature racine a une cardinalité de [1..1] et le mot-clé « opt » est équivalent à une cardinalité de [0..1].

  Lors de la construction de la table de symboles, le FeatureSymbol associé au feature racine se verra attribué les cardinalités de [1..1]. Un FeatureSymbol associé à un feature déclaré avec « opt » recevra les cardinalités de [0..1].

  Par défaut, un feature n'ayant pas de cardinalité explicite, ni de mot-clé « opt » recevra les cardinalités [1..1], tout comme le feature racine.

Cette solution permet de regrouper dans la table des symboles toutes les informations à propos des cardinalités de feature, quelque soit la manière dont ces cardinalités ont été déclarées (explicitement ou implicitement).

Ensuite, l'AST est parcouru une seconde fois afin d'associer les features partagés à leurs parents. En effet, lors du premier parcours, les features partagés sont associés à un seul de leurs parents, il s'agit du parent pour lequel le mot clé « shared » n'a pas été utilisé.

Un second parcours est alors nécessaire pour détecter les déclarations utilisant le mot-clé « shared » afin d'ajouter les liens entre les features partagés et leurs parents pour lesquels « shared » a été utilisé.

A la suite de ce second parcours, tous les FeatureSymbols de la table des symboles disposent de tous leurs liens vers leurs parents ainsi que des valeurs de cardinalités de feature.

Afin de vérifier la contrainte interdisant au features partagés d'être clonables, il suffit donc de vérifier qu'aucun feature avec plusieurs parents ne possède une cardinalité de feature maximale supérieure à 1.

5.6 Limites et travaux futurs

Comme cité lors de la présentation de ce chapitre, celui-ci n'a pas traité des conséquences de l'introduction du clonage sur le sous-ensemble de TVL permettant d'exprimer les contraites « cross-tree ».

La syntaxe et la sémantique des contraintes « cross-tree », ainsi que l'implémentation de ces contraintes dans le parser TVL, doivent être revues afin d'examiner les conséquences de l'introduction du clonage. Cette problématique étant complexe, le chapitre suivant y sera entièrement consacré.

Une autre limite de ce chapitre concerne l'implémentation de l'outil de faisabilité des modèles.

Le mémoire de P. Faber [24] présente l'utilisation d'un solver SAT [24,25] afin d'étudier la satisfaisabilité d'un modèle. Sans rentrer dans les détails de l'utilisation d'un solver, nous pouvons faire une analogie entre un tel solver et un moteur d'inférence tel que Prolog.

Le modèle TVL doit être traduit en un problème SAT, composé d'un ensemble de clauses logiques, telles que : $(a1 \vee a2) \wedge (a2 \vee \neg a3) \wedge \neg a4$

Le moteur d'inférence peut ensuite être interrogé en lui spécifiant des valeurs booléennes de variables, par exemple : $a1 = false \wedge a4 = false$

De façon analogue à l'interrogation d'un moteur d'inférence prolog, le solver peut ensuite rechercher s'il existe une solution respectant les clauses de sa base de connaissances et les conditions fournies lors de l'interrogation.

Comme décrit par P. Faber[24], l'implémentation du parser TVL antérieure à ce mémoire utilise un solver SAT permettant de vérifier la satisfaisabilité de l'ancienne version des contraintes « cross-tree ».

Un travail en cours de R. Michel consiste à définir un solver permettant de gérer le clonage.

Un travail futur pourrait donc consister à adapter l'outil de vérification de la satisfaisabilité en y intégrant un solver capable de gérer le clonage.

6. Révisions des contraintes.

Comme mentionné brièvement au chapitre 4, TVL permet d'exprimer des contraintes sur le modèle (cfr 4.2.4). Toutefois, l'introduction du clonage a un impact sur ces contraintes.

Ce chapitre va tout d'abord rappeler la syntaxe et la sémantique de ces contraintes.

Ensuite, il présentera les limites de ces syntaxe et sémantique dans le cadre du clonage.

Enfin, des adaptations de la syntaxe et de sa sémantique seront proposées afin de permettre l'utilisation de contraintes dans le cadre du clonage.

Toutefois, dans le cadre de ce mémoire, l'implémentation de cette nouvelle syntaxe et de sa sémantique n'a pas été réalisée.

6.1 Syntaxe et sémantique des contraintes de TVL

Comme vu au point 4.1, TVL dispose d'une syntaxe concrète définie par sa grammaire mais aussi d'une syntaxe abstraite dont la sémantique est définie formellement. Le langage abstrait décrit par la syntaxe abstraite et sa sémantique est noté $\mathcal{L}_{\mathbf{TVL}}$.

Une définition détaillée de la syntaxe et de la sémantique des contraintes de TVL, avant l'ajout des cardinalités de feature se trouve dans les documents de référence de TVL [19,20]. Cette section vise à en faire un bref rappel.

Un modèle exprimé selon la syntaxe concrète de TVL peut être normalisé, ce qui signifie qu'il est traduit sous une forme sémantiquement équivalente mais n'utilisant qu'un sous-ensemble de la syntaxe de TVL. Ce sous-ensemble de TVL dispose de notations sémantiquement équivalentes dans la syntaxe abstraite dont la sémantique est définie.

Ce principe s'applique également à la syntaxe des contraintes de TVL.

Les contraintes sont exprimées sous forme d'expressions, ces expressions peuvent également être normalisées afin de n'utiliser qu'un sous-ensemble de la syntaxe des expressions de TVL. Ce sous-ensemble est associé à des notations sémantiquement équivalentes dans le langage abstrait des expressions, noté $\mathcal{L}_{\mathbf{exp}}$.

Cette section va présenter brièvement la syntaxe abstraite des expressions, puis la sémantique associée à cette syntaxe et enfin la syntaxe concrète des expressions dans TVL.

6.1.1 Syntaxe abstraite et sémantique des expressions

Selon le document « Syntax & Semantic of TVL » [20], la syntaxe abstraite et sa sémantique sont définies comme ceci :

« $\mathcal{L}_{\mathbf{exp}}$ is the set of all correctly typed boolean expressions B, over the set F of Features and the set A of attributes, formed according to the expression grammar $G_{\mathbf{exp}}$, where f ∈ F is a feature, a ∈ A is an attribute, d ∈ Z is an integer, r ∈ R is a real, q ∈ Q is a rational number and t is an enum value. »

Voici la grammaire G_{exp} définissant la syntaxe des expressions dans ce langage abstrait (cfr « Syntax & Semantic of TVL » [20]):

B représente les expressions booléennes telles que : true/false, des attributs de type booléens, des features (la présence d'un feature est assimilée à true), des expressions booléennes avec opérateurs logiques (and, or, not, implication) ainsi que des expressions booléennes avec des opérateurs arithmétiques de comparaisons.

```
B ::=   true | false | f | a | E in S |
    f excludes f | f requires f |
    B && B | B || B | ! B |
    B -> B | B <- B | B <-> B |
    E == E | E != E |
    E <= E | E < E | E >= E | E > E |
    and(B[,B]*) | or(B[,B]*) | xor(B[,B]*)
```

E représente les expressions, telles que : les valeurs des attributs, les nombres, les expressions arithmétiques ainsi que des fonctions d'agrégations : addition, multiplication, minimum et maximum.

```
E ::=   a | t | d | q |
    E + E |
    E - E | E / E | E * E | - E |
    abs(E) | B ? E : E |
    sum(E[,E]*) | mul(E[,E]*) |
    min(E[,E]*) | max(E[,E]*)
```

S représente les listes d'expressions et les intervalles, c'est l'axiome de départ de la grammaire :

```
S ::= {E[,E]*} | [(d|*)..(d|*)] | [(r|*)..(r|*)]
```

Afin de déterminer la sémantique des expressions, une table de priorité et d'associativité des opérateurs est nécessaire. Les opérateurs y sont listés par priorité descendante.

OPERATOR PRECEDENCE IN TVL AND TVL_{NF}

Associativity	Operators
right	!, (unary) -, and aggregation functions
non	**requires, excludes**
left	*, /
left	+, -
non	>, <, >=, <=
non	==, ! =, **in**
left	**&&**
left	||
non	<->
left	->
right	<-
right	? :

Priorité des opérateurs et associativité (Extrait de [20])

Voici la définition sémantique des termes de la grammaire issue du document « Syntax & Semantic of TVL » [20] :

Quelques cas de cette sémantique pour illustration :

- la sémantique du mot-clé « true » est true (au sens logique).
- un nom de feature « n » signifie true si ce feature « n » est sélectionné dans le produit « c » sur lequel porte la contrainte.
- un identifiant d'attribut « a » signifie la valeur de l'attribut « a »
- and (B1, B2, B3) signifie true si les expressions B1,B2,B3 sont true

EXPRESSION SEMANTICS IN TVL AND TVL_{NF}

$$
\begin{aligned}
[\![\textbf{true}]\!] &= true \\
[\![\textbf{false}]\!] &= false \\
[\![\text{n}]\!] &= true \text{ iff } n \in c \\
[\![\text{a}]\!] &= v(a) \\
[\![\text{a == t}]\!] &= true \text{ iff } v(a) \text{ equals } t \\
[\![\text{a != t}]\!] &= true \text{ iff } v(a) \text{ does not equal } t \\
[\![\text{E } \textbf{in} \text{ S}]\!] &= true \text{ iff } [\![\text{E}]\!] \in [\![\text{S}]\!] \\
[\![\text{n}_1 \textbf{ requires } \text{n}_2]\!] &= true \text{ iff } n_1 \notin c \vee n_2 \in c \\
[\![\text{n}_1 \textbf{ excludes } \text{n}_2]\!] &= true \text{ iff } n_1 \notin c \vee n_2 \notin c \\
[\![\text{B}_1 \text{ \&\& } \text{B}_2]\!] &= true \text{ iff } [\![\text{B}_1]\!] \wedge [\![\text{B}_2]\!] \\
[\![\text{B}_1 \text{ || } \text{B}_2]\!] &= true \text{ iff } [\![\text{B}_1]\!] \vee [\![\text{B}_2]\!] \\
[\![\text{!B}]\!] &= true \text{ iff } [\![\text{B}_1]\!] \text{ equals } false \\
[\![\text{B}_1 \text{ -> } \text{B}_2]\!] &= true \text{ iff } [\![\text{B}_1]\!] \implies [\![\text{B}_2]\!] \\
[\![\text{B}_1 \text{ <- } \text{B}_2]\!] &= true \text{ iff } [\![\text{B}_2]\!] \implies [\![\text{B}_1]\!] \\
[\![\text{B}_1 \text{ <-> } \text{B}_2]\!] &= true \text{ iff } [\![\text{B}_1]\!] \Leftrightarrow [\![\text{B}_2]\!] \\
[\![\text{E}_1 == \text{E}_2]\!] &= true \text{ iff } [\![\text{E}_1]\!] \text{ equals } [\![\text{E}_2]\!] \\
[\![\text{E}_1 \text{ != } \text{E}_2]\!] &= true \text{ iff } [\![\text{E}_1]\!] \text{ does not equal } [\![\text{E}_2]\!] \\
[\![\text{E}_1 < \text{E}_2]\!] &= true \text{ iff } [\![\text{E}_1]\!] < [\![\text{E}_2]\!] \\
[\![\text{E}_1 > \text{E}_2]\!] &= true \text{ iff } [\![\text{E}_1]\!] > [\![\text{E}_2]\!] \\
[\![\text{E}_1 \text{ <= } \text{E}_2]\!] &= true \text{ iff } [\![\text{E}_1]\!] \leq [\![\text{E}_2]\!] \\
[\![\text{E}_1 \text{ >= } \text{E}_2]\!] &= true \text{ iff } [\![\text{E}_1]\!] \geq [\![\text{E}_2]\!] \\
[\![\textbf{and}(\text{B}_1, .., \text{B}_k)]\!] &= true \text{ iff } \textstyle\bigwedge_{i \in [1,k]} [\![B_i]\!] \\
[\![\textbf{or}(\text{B}_1, .., \text{B}_k)]\!] &= true \text{ iff } \textstyle\bigvee_{i \in [1,k]} [\![B_i]\!] \\
[\![\textbf{xor}(\text{B}_1, .., \text{B}_k)]\!] &= true \text{ iff } \textstyle\bigoplus_{i \in [1,k]} [\![B_i]\!] \\
[\![\text{E}_1 \text{ + } \text{E}_2]\!] &= \text{the value of } [\![\text{E}_1]\!] + [\![\text{E}_2]\!] \\
[\![\text{E}_1 \text{ - } \text{E}_2]\!] &= \text{the value of } [\![\text{E}_1]\!] - [\![\text{E}_2]\!] \\
[\![\text{E}_1 \text{ / } \text{E}_2]\!] &= \text{the value of } [\![\text{E}_1]\!] \ / \ [\![\text{E}_2]\!] \\
[\![\text{E}_1 \text{ * } \text{E}_2]\!] &= \text{the value of } [\![\text{E}_1]\!] * [\![\text{E}_2]\!] \\
[\![\text{-E}]\!] &= \text{the value of } -[\![\text{E}]\!] \\
[\![\textbf{abs}(\text{E})]\!] &= \text{the absolute value of } [\![\text{E}]\!] \\
[\![\text{B}_1 \text{ ? } \text{E}_1 \text{ : } \text{E}_2]\!] &= \text{if } [\![\text{B}_1]\!], \text{ then } [\![\text{E}_1]\!], \text{ otherwise } [\![\text{E}_2]\!] \\
[\![\textbf{sum}(\text{E}_1, .., \text{E}_k)]\!] &= \text{the value of } \textstyle\sum_{i \in [1,k]} [\![E_i]\!] \\
[\![\textbf{mul}(\text{E}_1, .., \text{E}_k)]\!] &= \text{the value of } \textstyle\prod_{i \in [1,k]} [\![E_i]\!] \\
[\![\textbf{min}(\text{E}_1, .., \text{E}_k)]\!] &= \text{the smallest value of the } [\![E_i]\!] \\
[\![\textbf{min}(\text{E}_1, .., \text{E}_k)]\!] &= \text{the greatest value of the } [\![E_i]\!] \\
[\![\{ \text{E}_1, .., \text{E}_k \}]\!] &= \text{the set } \{[\![E_i]\!] | i \in [1,k]\} \\
[\![[\text{d}_1 \text{ .. } \text{d}_2]]\!] &= \text{the interval } [d_1, d_2] \\
[\![[\text{ * .. d }]]\!] &= \text{the interval }]-\infty, d] \\
[\![[\text{d .. * }]]\!] &= \text{the interval } [d, +\infty[\\
[\![[\text{f}_1 \text{ .. } \text{f}_2]]\!] &= \text{the interval } [f_1, f_2] \\
[\![[\text{ * .. f }]]\!] &= \text{the interval }]-\infty, f] \\
[\![[\text{f .. * }]]\!] &= \text{the interval } [f, +\infty[
\end{aligned}
$$

FIGURE 6.1 Sémantique des expressions de TVL

6.1.2 Syntaxe concrète des expressions

Comme expliqué précédemment, les éléments de la syntaxe abstraite correspondent aux éléments d'un sous-ensemble de la syntaxe concrète.

Ce sous-ensemble décrit la syntaxe de la version normalisée de TVL. TVL dispose d'autres éléments supplémentaires utiles afin de fournir un meilleur confort d'utilisation du langage mais qui n'apportent rien de plus au niveau sémantique.

Des expressions TVL exprimées avec les éléments de TVL peuvent donc être traduites dans la version normalisée de TVL, celle-ci ayant une correspondance avec la syntaxe abstraite dont la sémantique est définie au point précédent.

La définition de la syntaxe concrète des expressions de TVL se trouve dans le document « Syntax & Semantic of TVL » [20] (p7). Les principes permettant de traduire un modèle TVL vers sa version normalisée sont décrits dans ce même document ([20], p10).

La syntaxe concrète offre de nouvelles constructions, par exemple :

- **children** permet de représenter tous les sous-feature d'un feature, en le combinant avec une fonction d'agrégation telle que max, cela permet d'écrire max(children.a) afin de rechercher le maximum parmi les valeurs de l'attribut « a » au lieu d'écrire max (c1.a, c2.a,…) où les ci sont les sous-features du feature courant.
- **Selectedchildren** est similaire à children mais ne prend en compte que les sous-features sélectionnés dans le produit pour lequel on évalue les contraintes.

La normalisation consiste à exprimer les contraintes comportant des éléments de TVL non repris dans sa forme normalisée, en contraintes de sémantique similaire en n'utilisant que des éléments de la forme normalisée, par exemple, traduire le max(children.a) en max (c1.a, c2.a, c3.a) où c1,c2,c3 sont les sous-features.

6.2 Limites de la syntaxe et sémantique des expressions

La syntaxe abstraite référence des features et des attributs (cfr 6.1.1), mais comment interpréter ces références vu l'introduction des clones ?

Par exemple, considérons l'exemple suivant issu du document de Michel et al. [22] à propos de la syntaxe et la sémantique des cardinalités de feature (et donc du clonage) :

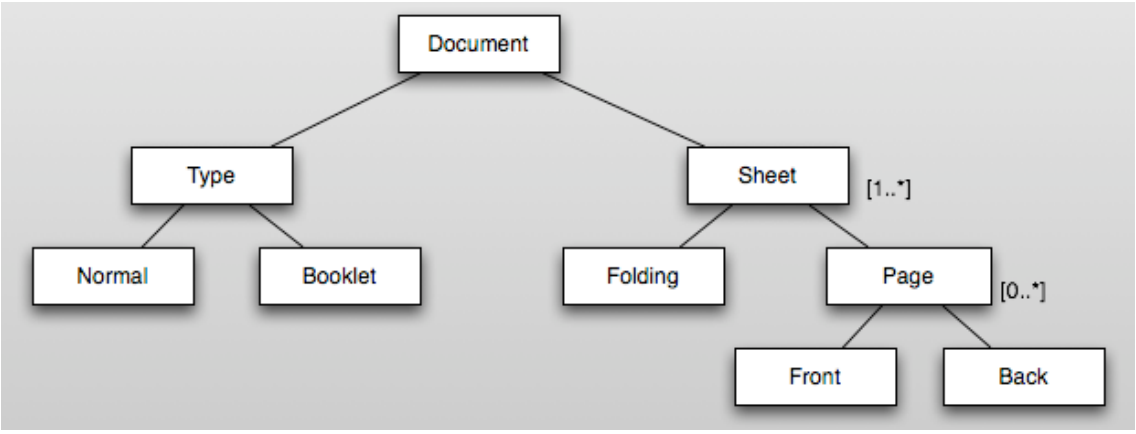


Figure 6.2 Feature diagram « Document »

Dans le modèle décrit à la figure 6.2, que signifie la contrainte **Booklet -> Folding** ?

Dans la version de TVL sans clonage, cette contrainte signifiait que si le feature « Booklet » était sélectionné, « Front » devait l'être également.

Mais suite à l'introduction du clonage, que signifie cette contrainte si un produit comporte plusieurs clones de « Sheet » ? La présence de « Booklet » implique-t-elle la présence de « Folding » sous tous les clones de « Sheet », ou un seul des clones, ou un certains nombre de clones ?

La réponse à cette question ne peut pas être obtenue avec l'ancienne syntaxe des expressions puisqu'elle dépend de la signification du modèle dans le monde réel.

Intuitivement, si un document est de type « Booklet », on pourrait supposer que toutes ses feuilles devraient être de types « Folding », mais dans le cas d'une autre contrainte, par exemple **Normal -> Front,** on pourrait supposer qu'au moins une des pages devrait être de type « Front ».

Le choix de la portée de la contrainte, sur au moins un des clones ou sur tous les clones ne peut donc pas se faire implicitement.

Bien entendu, d'autres cas posent problèmes, par exemple lors de l'utilisation de la valeur d'un attribut, chaque clone peut disposer d'une valeur de cet attribut, alors quelle valeur utiliser, celle d'un clone en particulier, la valeur moyenne ou totale de tous les clones ?

Enfin, l'introduction du clonage peut entrainer de nouvelles contraintes à propos du clonage, par exemple le nombre de clones de deux features « A » et « B » doivent être identiques.

Vu le grand nombre de cas possibles et de leurs combinaisons, la résolution de ces problèmes ne doit pas se faire via une approche au cas par cas, mais via une approche plus globale.

6.3 Propositions d'adaptations de la syntaxe

La méthode utilisée dans le but d'élaborer les modifications à apporter à la syntaxe consiste à partir des éléments de la syntaxe abstraite et de compléter celle-ci.

Les éléments de cette syntaxe devenus ambigus suite à l'introduction du clonage sont adaptés ou remplacés par de nouveaux éléments dont la sémantique est précisée. Ces modifications sont alors transposées à la syntaxe concrète de TVL.

Ensuite les « sucres syntaxiques » de TVL n'appartenant pas à la version normalisée de TVL sont adaptés suite à l'introduction du clonage et des modifications opérées sur la syntaxe abstraite et sa sémantique.

Les modifications de ces sucres syntaxiques peuvent engendrer des adaptations dans la syntaxe abstraite si la nouvelle version de celle-ci, élaborée à partir de l'ancienne version, ne permet pas de les intégrer. Toutefois, pour faciliter la lecture, ces nouveaux éléments de la syntaxe abstraite seront décrits lors de la présentation des modifications de la syntaxe abstraite.

Les points suivants présentent donc les modifications de la syntaxe abstraite et la sémantique associée, puis la transposition de ces éléments à la syntaxe concrète.

6.3.1 Feature

La grammaire abstraite des expressions (cfr.6.1.1) permet d'utiliser des noms de features. La sémantique du nom du feature signifie true si ce feature est sélectionné dans le produit.

Mais comme illustré au point 6.2, ceci est ambigu suite à l'introduction du clonage. Afin de lever cette ambiguïté, il est nécessaire de préciser si l'expression composée d'un nom de feature est true si un clone existe dans le produit ou si toutes les instances de feature pouvant sélectionner ce sous-feature le sélectionne.

Cette précision ne pouvant pas se faire implicitement puisque le choix dépend du contexte réel représenté par le produit, ce choix doit être spécifié par le langage. L'introduction de quantificateurs est donc nécessaire.

Quantificateurs :

Les mathématiques nous fournissent deux quantificateurs :

- ∀ x : C signifiant que pour tout x, la condition C(x) doit être vérifiée
- ∃ x : C signifiant qu'il existe au moins un x tel que C(x) est vérifié

OCL [28] fournit également ces quantificateurs, applicables à des collections d'objets, sous la forme :

- collection->forall(x : Type | C(x)) : vrai si tous les éléments de la collection vérifie C
- collection->exists(x : Type | C(x)) : vrai si au moins un élément de la collection vérifie C

où :

- collection est une collection d'objets
- x est l'élément courant de la collection
- Type est le type de l'objet
- C(x) est une expression OCL contenant la variable x

Application à la syntaxe abstraite :

Ces quantificateurs permettent de résoudre le problème des noms de features de TVL, la syntaxe abstraite (cfr 6.1.1) va donc être adaptée afin d'intégrer ces quantificateurs.

- suppression de **B ::= f** puisque son utilisation est ambiguë.
- ajout du quantificateur « forAll »
- ajout du quantificateur « exists »

Reprenons l'exemple de la figure 6.2, afin d'illustrer ces quantificateurs. La contrainte : *Si un document est de type « Booklet », toutes ses « Sheets » doivent être de type « Folding »* peut s'exprimer de la sorte, de façon intuitive :

```
Exists(Booklet) -> forAll(Sheet)
                    {
                        exists(Folding)
                    }
```

Par rapport à l'ancienne contrainte « Booklet -> Folding », le feature « Sheet » fait son apparition. Cela est nécessaire afin de préciser les choix effectués à propos des features clonables : Rechercher à valider une condition sur tous ses clones descendants ou en rechercher un seul validant la condition.

Dans le cas de la figure 6.2, en parcourant le modèle au départ de la racine, le feature clonable « Sheet » est rencontré avant d'atteindre le feature « Booklet ». Le choix effectué concernant les « Sheet » doit donc être spécifié.

La condition « exists(Folding) », exprimée dans le corps du « forAll(Sheet) » sera donc vérifiée pour toutes les occurrences de « Sheet ».

Enchainement des qualificateurs :

Ces quantificateurs peuvent s'enchainer afin d'exprimer des conditions sur des nœuds de différentes profondeurs dans le graphe constitué par le modèle.

Le chainage des quantificateurs s'opère donc de façon similaire à celui de ∀ et ∃ en mathématique.

Par exemple : *Si un Document est de type « Normal », il existe au moins une « Page » de type « Front », indépendamment des Sheets.*

```
Exists(Normal) -> exists(Sheet) {
                      exists(Page) {
                            exists(Front)
                      }
                  }
```

Autre exemple un peu différent : *Si un Document est de type « Normal », il existe au moins une « Page » de type « Front », **sous chaque « Sheet »**.*

```
Exists(Normal) -> forAll(Sheet) {
                      exists(Page) {
                            exists(Front)
                      }
                   }
```

Cette condition est vérifiée si pour chaque « Sheet » il existe une « Page » sous laquelle il existe un « Front », ce qui est équivalent à dire que pour chaque « Sheet », il existe un front. La contrainte peut donc s'exprimer de la sorte :

```
Exists(Normal) -> forAll(Sheet) {
                      exists(Front)
                  }
```

Une condition « exists » est vérifiée si un des sous-arbres du clone courant vérifie la condition. Donc si l'on recherche l'existence d'un clone vérifiant la condition et descendant du clone courant, il n'est pas nécessaire de spécifier un « exists » à chaque niveau de feature clonable rencontré dans l'arbre, les « exists » intermédiares entre le dernier « forAll » et le dernier « exists » qui défini l'élément à rechercher au plus bas niveau sont donc facultatifs.

Grammaire de la syntaxe abstraite :

L'introduction des quantificateurs est formalisée dans la grammaire définissant la syntaxe abstraite.

Reprenons les règles de production des expressions booléennes de la syntaxe abstraite (pour rappel, « f » est un feature et « a » un attribut) :

```
B ::=   true | false | f | a | E in S |
    f excludes f | f requires f |
    B && B | B || B | ! B |
    B -> B | B <- B | B <-> B |
    E == E | E != E |
    E <= E | E < E | E >= E | E > E |
    and(B[,B]*) | or(B[,B]*) | xor(B[,B]*)
```

La règle de production « B ::= f » n'est plus autorisée car ambigüe, elle est remplacée par des règles utilisant les quantificateurs.

La règle de production pour le « forAll » est la suivante :
`B ::= forAll(f) {B}`

Et pour « exists » :
`B ::= exists(f) {B}`

Sémantique de forAll et exists :

Suite à l'imbrication des « forAll » et « exists », la définition est récursive. Pour rappel, « f » est un nom de feature et « B » est une expression booléenne.

Soit « pj » un clone présent dans le produit à vérifier.

Soit le Set « C » composé des clones du feature « f », descendant de « pj » :

[[forAll (f) {B}]] = true iff $\forall c \in C$: [[B]] = true
[[exists (f) {B}]] = true iff $\exists c \in C$: [[B]] = true

Appelons le forAll/exists décrit ci-dessus le « quantificateur de base ».

Appelons « quantificateurs intégrés » les « forAll » et « exists » éventuellement présents dans l'expression « B » du corps du quantificateur de base.

Pour chaque quantificateur intégré, son clone « pj » est un des clones ci du quantificateur de base.

S'il n'y a pas de quantificateur de base, « pj » référence l'élément racine (root) du produit.

Cette définition implique que le point de départ de la définition des quantificateurs est toujours à la racine de l'arbre. La contrainte est donc définie de la même manière qu'elle soit définie au niveau de la racine ou en profondeur dans l'arbre.

6.3.2 Excludes et Requires

Dans l'ancienne version de la syntaxe abstraite, « F1 excludes F2 » signifiait que le si le feature « F1 » était présent dans le produit, « F2 » ne pouvait pas l'être.

Intuitivement, « excludes » pourrait être adapté au clonage. « F1 excludes F2 » signifierait alors que si le produit contient un clone de « F1 », il ne peut pas contenir de clone de « F2 ».

Mais si « F1 » et « F2 » possède un parent commun « P » clonable, l'exclusion a-t-elle une portée globale ou locale sous chaque clone de « P » ?

« Excludes » est donc ambigu, mais l'utilisation d'implication booléenne (->) et de quantificateurs permet de se passer du mot-clé « excludes ».

En effet, « F1 excludes F2 » est sémantiquement équivalent à « F1 -> !F2 ». « F1 excludes F2 » est vrai si « F1 » n'est pas sélectionné par le produit ou si « F2 » n'est pas sélectionné par ce produit (cfr figure 6.1).

Or, en logique, « x -> y » est équivalent à « ¬x ∨ y ». Remplaçons « x », « y » par « F1 », « ¬F2 » et nous obtenons « ¬F1 ∨ ¬F2 ».

La table de vérité de « F1 excludes F2 » est donc équivalente à celle de « F1 -> !F2 »

F1	F2	¬F1 ∨ ¬F2	F1 exclude F2
1	1	0	0
1	0	1	1
0	1	1	1
0	0	1	1

Selon le même principe, « F1 requires F2 » est vrai si « F1 » n'appartient pas au produit ou si « F2 » appartient au produit. Ce qui par analogie au cas de « excludes » est équivalent à « ¬F1 ∨ F2 », qui est donc équivalent à « F1 -> F2 ».

Par exemple, si la présence de « F1 » exclu la présence du « F2 » localement sous chaque parent « P » :

```
forAll(P) {
    exists(F1) -> ! exists(F2)
}
```

Autre exemple, si la présence d'un clone de « F1 » sous n'importe quel clone de « P » exclue la présence d'un clone de « F2 » sous n'importe quel clone de « F2 » :

```
Exists(P){ exists(F1) } -> ! exists(P){exists(F2)}
```

Même principe avec « F1 requires F2 », sémantiquement équivalent à « F1 -> F2 ».

6.3.3 Attributs et set de features

Comme vu au point précédent, l'ancienne version de la syntaxe abstraite permet l'utilisation d'attributs. La sémantique d'un attribut « a » est la valeur de cet attribut « a ».

Mais dans le cas d'existence de plusieurs clones du feature propriétaire de cet attribut, la sémantique doit être adaptée.

Description intuitive

Afin d'illustrer ce cas considérons un feature diagram dont la racine est « R », composé d'un feature clonable « X », lui-même constitué de feature clonables « A » et « B » composés respectivement des features clonables « B » et « Q ». Ces features « B » et « Q » disposent d'un attribut « value ».

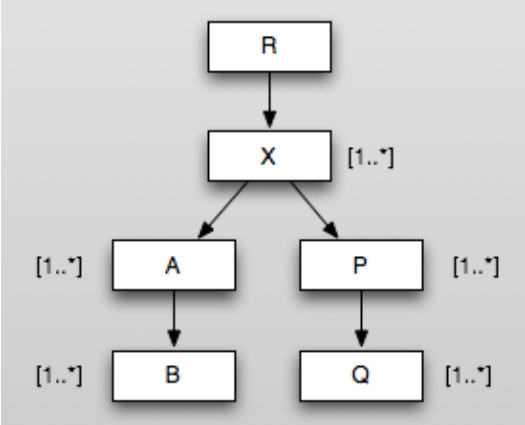


Figure 6.1 – Modèle avec features clonables

Voici un exemple de produit issu de ce modèle :

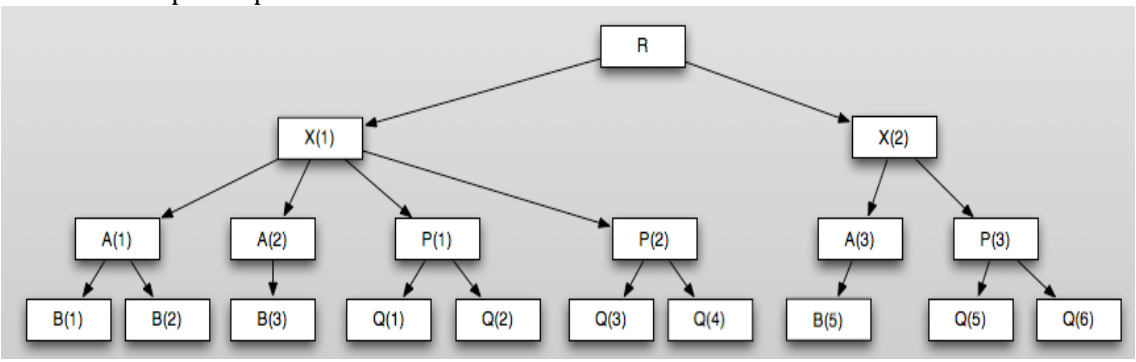


Figure 6.2 - Produit issu du modèle de 6.1

Remarque : *Dans ce schéma 6.2, les notations telles que B(1), B(2) permettent de représenter des clones particuliers du schéma uniquement afin de clarifier les explications. Mais TVL ne dispose pas de moyen de donner un identifiant à un clone en particulier, bien que nous verrons qu'il est possible de sélectionner un ensemble de clones (cfr 6.3.5).*

Considérons la contrainte suivante :

```
forAll(X){  forAll(A) {
                « corps du forAll »
        } }
```

Quelle serait la signification de « B.value » dans le corps du forAll ? Une solution intuitive est de considérer qu'il s'agit de la collection des valeurs des attributs « value » des clones du feature « B ».

Une autre question se pose alors, doit-on considérer les clones localement sous un parent ou globalement ? La solution locale est retenue car elle offre plus de flexibilité.

En effet, dans l'exemple précédent, la contrainte booléenne exprimée dans le corps du « forAll » sera vérifiée pour chaque clone de « A ».

Par exemple, dans le cadre de la vérification de la contrainte sous le clone A(1), « B.value » signifie la collection {B(1).value, B(2).value}.

Dans le cadre de la vérification pour A(2), la collection est {B(3).value}.

Choisir la solution globale aurait engendré une collection identique dans tous les cas, composé de {B(1).value, B(2).value, B(3).value, B(4).value }, ce qui est indépendant du contexte. Une collection dépendante du contexte n'aurait pas été permise.

Par contre, la solution locale permet également de générer un Set contenant des clones d'un niveau plus global. Il suffit de spécifier que ce Set est construit à partir d'un ancêtre du clone courant (appelé « **clone commun** »), ce qui permettra d'y inclure des clones « cousins » ou « petits cousins ».

Par exemple, toujours dans le cadre de la vérification de la contrainte pour A(1), où A(1) est appelé clone courant, « X.B.value » indiquerait que le clone commun est un clone de « X », ce qui permettrait de remonter dans l'arbre et de générer le Set {B(1).value , B(2).value, B(3).value} alors que « R.B.value » remonterait jusqu'à la racine et génèrerait le Set {B(1).value, B(2).value, B(3).value, B(4).value}.

Syntaxe abstraite :

La description intuitive montre qu'il sera parfois nécessaire de qualifier un attribut, c'est-à-dire indiquer à quel feature il appartient et quel est le type de feature du clone commun recherché. Les features descendants du feature correspondant à l'ancêtre commun et ancêtres du feature propriétaire de l'attribut peuvent aussi être indiqués.

La règle SET_CLONE permet déterminer le Set des clones desquels extraire la valeur de l'attribut afin de constituer la collection de valeurs.
La règle QUALIFIED se base donc sur SET_CLONE, auquel le nom de l'attribut à utiliser est ajouté.

```
SET_CLONE ::= [( p «.» )*] f
QUALIFIED ::= SET_CLONE «.» a
```

- « » représente un littéral, [] un élément facultatif et ()* une répétition de 1 à n fois des éléments entre ().
- (p « . »)* représente une suite de noms de features : {p1, … pk} avec k >0.
- p1 est le feature dont le « parent commun » est un clone.
- Les p2,..,pk sont descendants de p1 et ancêtres de f, tel que pi+1 est descendant de pi.
- f est le feature propriétaire de l'attribut et a l'attribut.

Dans la règle SET_CLONE, les « p » peuvent être omis, le clone courant est alors considéré comme clone commun. Ce qui implique que les clones recherchés afin de constituer la collection de valeurs seront des clones descendants du clone courant, « f » doit donc dans ce cas être descendant du clone courant.

Remarque : Dans la suite de ce mémoire, l'acronyme « p.f.a » désignera les notations permettant de constituer un ensemble de valeurs à partir d'un Set de clones, comme décrit ci-dessus.

Si l'expression d'un attribut qualifié est utilisée hors du corps d'un quantificateur (forAll/exists), le clone courant est alors le clone du feature racine.

Les attributs qualifiés pourront être utilisés en tant qu'expression à part entière uniquement si la collection d'attributs qu'ils engendrent ne contient qu'une seule valeur. Si la collection contient plusieurs valeurs, celle-ci pourra uniquement être utilisée comme argument de fonctions d'agrégations ou de count.

Dans l'ancienne version de la syntaxe et de sa sémantique, la règle de production « E ::= a » représentait un attribut, qualifié ou non. Vu l'introduction du clonage et la nécessité de distinguer les différents features cités dans l'identifiant qualifié d'un attribut, la règle « E ::= a » sera donc permise mais le « a » représentera un attribut non qualifié.

Cette règle est donc complétée afin de permettre l'utilisation d'attributs qualifiés en tant qu'expression : « E ::= QUALIFIED | a »

Définition Sémantique :

Conformément à la description intuitive, la définition sémantique de la présence d'un attribut dans une expression est adaptée.

La définition utilise la notion de descendant, définie récursivement :
Le feature Y est descendant de X $\Leftrightarrow$ (X,Y) $\in$ DE $\vee$ ($\exists$P : (P,Y) $\wedge$ P descendant de X)

Voici la sémantique des attributs (qualifiés ou non) :
Soit SX = {x1, x2, … xk} un Set de clones d'un feature X pour lequel l'on souhaite vérifier la condition booléenne B (dans le cadre d'un « forAll(X) {B} » ou d'un « exists(X) {B} »).

$\forall$ xi $\in$ SW :

--Définition de « f.a »
Soit SY = {y1,y2,…yn} un Set de clones d'un feature Y, descendant de X tq :
$\forall$ yi $\in$ SY : yi est descendant de xi
Si n > 1 : [[Y.a]] = {[[y1.a]], [[y2.a]], …, [[yn.a]]}
Sinon [[Y.a]] = [[y1.a]]

--Définition de « p.f.a »
Soit pk un clone d'un feature P, tel que xi est descendant de pk,
Soit SQ = {q1, … qm} les clones de Q descendants de pk,
Alors si m > 1 [[P.Q.a]] = {[[q1.a]], …, [[qm.a]]}
Sinon [[P.Q.a]] = [[q1.a]]

--Définition de «a»
[[a]] = [[xi.a]]

Remarques :

- La condition B est vérifiée pour un clone « c » si son évaluation dans le contexte de du clone « c » (i.e. évaluation des QUALIFIED, exists et allOf avec « c » comme clone courant) est vraie.
- la sémantique de « p.f.a » est identique à celle de « p*.f.a », les features autres que les premier et dernier ne jouant ici qu'un rôle indicatif (Nous verrons en 6.3.6 qu'ils peuvent avoir une autre utilité).
- [[clone.attribute]] signifie la valeur de l'attribut « attribute » du clone « clone »

6.3.4 Fonctions d'agrégation

L'ancienne version de la syntaxe permettait l'utilisation de fonctions d'agrégation telles que « sum », « mul », « min » et « max » sous la forme « fct(E, [E*]) » où E est une expression et « fct » est une des 4 fonctions d'agrégation.

Vu l'adaptation de la sémantique d'un attribut dans une expression, la sémantique des fonctions d'agrégation doit également être adaptée afin de pouvoir manipuler des collections de valeurs issues d'attributs.

Illustrons cette modification par un exemple : l'expression « sum(X.price, Y.price) » signifiait la somme des attributs « price » des features X et Y. Mais comme nous l'avons vu au point 6.2.3, « X.price » et « Y.price » sont des collections pouvant contenir plusieurs valeurs, par exemple {x1.price, x2.price} et {y1.price}. La sémantique de l'expression « sum(X.price, Y.price) » devient donc celle de « sum(x1.price, x2.price, y1.price) ». Bien entendu, il en va de même pour les autres fonctions d'agrégation.

Syntaxe abstraite

La règle est toujours la suivante :

```
E ::= mul(E, [E*]) | sum(E, [E*]) | max(E, [E*]) | min(E, [E*]);
```

Mais comme nous l'avons vu au point précédent, E peut être un attribut qualifié représentant une collection de valeurs.

Définition Sémantique

L'ancienne définition sémantique était la suivante :

[[sum(E1,… Ek)]] = value of $\Sigma_{i \in [1..k]}$ [[Ei]]

La nouvelle définition, dans laquelle une fonction d'agrégation peut recevoir en argument des attributs représentant des collections de valeurs est la suivante :

[[sum(X1,… Xk)]] = valeur de $\Sigma_{i \in [1..k]}$ [[Vi]]

où :

Si Xi est de la forme QUALIFIED

alors Vi = sum(a1, … ak) où {a1,…ak} est la collection des valeurs associées à « a »(cfr 6.2.3 définition sémantique)

Sinon Vi = Xi

6.3.5 Filtre et valeurs calculées des collections

Les points précédents permettent de résoudre les ambiguïtés sémantiques causées par la présence de plusieurs clones d'un feature lors de l'utilisation de noms de features et d'attributs dans les expressions.

En résumé, ce qui autrefois représentait une valeur représente une collection de valeurs issues des différents clones.

Les sections précédentes déterminent comment constituer ce set de clones, cela dépend du contexte d'utilisation (cfr 6.2.1), le set est constitué de tous les clones descendants d'un clone ancêtre commun.

Cette section vise à présenter quelques améliorations de l'expressivité du langage afin d'offrir plus de possibilités lors de la constitution du set de clones.

Filtre

Une première proposition consiste à fournir un filtre permettant de ne retenir parmi les clones associés au contexte que les clones qui vérifient une condition.

Cette amélioration permettrait par exemple de considérer uniquement certains clones de sous-features lors du calcul d'une somme.

Dans la version précédente, un feature avait au maximum une instance dans un produit. Indiquer quels features utilisés dans la somme permettait donc de choisir quelles valeurs d'attributs reprendre. Mais puisqu'un feature peut avoir plusieurs clones, spécifier quels features prendre en compte n'est pas suffisant, il est nécessaire de spécifier quels clones considérer dans la somme.

Vu l'utilisation de collection afin de représenter les clones et leurs valeurs, sélectionner les clones selon un numéro d'ordre n'est pas possible, puisqu'ils ne sont pas ordonnés. De plus, aucune notion d'ordre n'existe dans TVL pour les sous-features d'un feature, ni pour ses attributs.

Un filtre va donc être utilisé afin de sélectionner quels clones prendre en compte. Reprenons l'exemple du point 6.2.3 afin d'illustrer son utilisation.

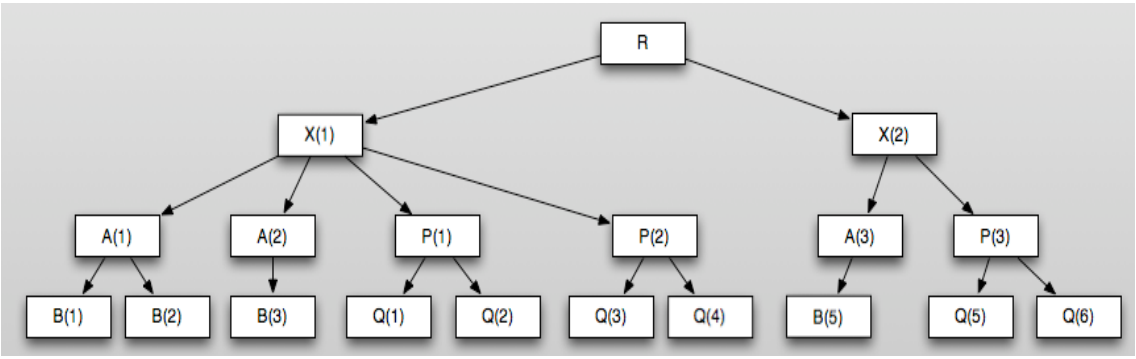


Figure 6.2

Ce produit est constitué d'une instance d'un feature racine R et de clones des features X,A,B,P et Q. B,P et Q disposent d'un attribut « price » de type entier.

Considérons l'expression :

```
forAll(X) { forAll(A) {
      //corps de la condition
}}
```

Nous avons vu que l'expression ci-dessus permet de vérifier une condition exprimée dans le « corps de condition » pour chaque clone de A.

Pour rappel, dans le contexte de la vérification pour le clone A(1), « B.price » représente la collection {B(1).price, B(2).price}. « X.Q.price » représente la collection {Q(1).price, Q(2).price, Q(3).price, Q(4).price } (De façon imagée : on remonte de A(1) vers un clone de X, X(1) est trouvé, on redescend alors sous X(1) afin de sélectionner les clones de Q)

Conformément à la règle syntaxique PFA, « X.Q.price » a été utilisé, le 1er nom de feature indique le type du feature parent à rechercher (X). Le dernier nom de feature indique le type propriétaire de l'attribut (Q). Le feature P a été omis, mais la règle « p.f.a » pourrait être adaptée afin de citer les différents features présents entre « p » et « f », ce qui donnerait dans l'exemple : « X.P.Q.price » (sémantiquement équivalent à X.Q.price).

Un filtre pourrait alors intervenir sur chaque feature clonable afin de choisir quels clones retenir, sauf pour le 1er, puisque l'on ne considère que l'instance de ce feature (X(1)) rencontrée en remontant l'arbre à partir du clone courant (ex : A(1)).

Une contrainte pourrait être : *Pour tout clone de A, la somme des attributs « price » des clones de B attachés à ce clone de A doit être supérieure à la somme des « Q.price » attachés au clone de P ayant la plus grande valeur de l'attribut « price », parmi les clones de P attachés à un clone de X auquel le clone de A est aussi attaché*.

« X.P.Q.price » ne convient pas à cet exemple, il convient de filtrer les P retenus afin de ne retenir que celui ayant le « price » maximum, la contrainte s'exprime alors de la façon suivante :

```
forAll(X) { forAll(A){
           sum(B.price) > sum(X.P[filtre].Q.price)
}}
```

où *filtre* est une expression booléenne B évaluée à true uniquement pour le clone de P ayant le « price » maximum :

```
price = max(X.P.Price)
```

La condition de *filtre* est évaluée pour chaque occurrence de P descendante du X de départ, le contexte lors de l'évaluation du filtre est donc le clone de P pour lequel la filtre est vérifié. « price » dans cette condition se rapporte donc à l'attribut « price » du clone de P évalué, alors que « X.P.price » à partir de ce contexte remonte dans l'arbre jusqu'au clone de X commun et redescend donc afin de prendre toutes les valeurs de « price » rencontrées dans les clones de P sous le X commun. La condition est donc vraie pour le P ayant la valeur maximale (Nous faisons l'hypothèse ici chaque P du produit a une valeur de « price » différente).

La syntaxe d'un attribut qualifié deviendrait donc :

```
SET_CLONE  ::= ( p [«[»  B «]»] «.» )* f [«[»  B «]»]
QUALIFIED ::= SET_CLONE « . » a
```

où :

- [x] représente un élément facultatif x
- « x » signifie que x est un litéral
- * signifie que x est répété 1 ou plusieurs fois

Remarques :

- le feature f peut recevoir une condition
- les features de la suite de feature p* peuvent aussi recevoir une condition

La sémantique est basée sur la définition de la sémantique des attributs qualifiés (cfr 6.2.3), pour rappel :

Soit SX = {x1, x2, … xk} un set de clones d'un feature X pour lequel l'on souhaite vérifier la condition booléenne B (dans le cadre d'un forAll(X) {B} ou exists(X) {B}).

∀ xi ∈ SW :

--Définition de « f.a »
Soit SY = {y1,y2,…yn} un Set de clones d'un feature Y, descendant de X tq :
∀ yi ∈ SY : yi est descendant de xi
Si n > 1 : [[Y.a]] = {[[y1.a]], [[y2.a]], …, [[yn.a]]} si
Sinon [[Y.a]] = [[y1.a]]

--Définition de « p.f.a »
Soit pk un clone d'un feature P, tel que xi est descendant de pk,
Soit SQ = {q1, … qm} les clones de Q descendants de pk,
Alors si m > 1 [[P.Q.a]] = {[[q1.a]], …, [[qm.a]]}
Sinon [[P.Q.a]] = [[q1.a]]

--Définition de «a»
[[a]] = [[xi.a]]

L'introduction d'une condition booléenne Bq sur Q à pour conséquence d'exclure de SQ les qi pour lesquels Bq n'est pas respectée.

Contrairement à la sémantique des attributs qualifiés sans filtre, les features avec filtres, cités entre le 1er et le dernier, ont plus qu'un rôle indicatif.

Soit l'expression « P.K[Bk].Q[Bq] », nous avons vu que le set SQ est constitué des clones de Q, descendants de P et respectant la condition Bq. Mais ces clones de Q sont des descendants de clones de K, eux-mêmes descendants du clone commun de P. La condition Bk réduit le set constitué de clones de K, les clones de Q descendants des clones de K exclus par Bk sont alors exclus de SQ, même s'ils respectent Bq. Les filtres fonctionnent donc en cascades.

Expressions calculées

Une seconde proposition est de permettre de ne pas constituer la collection à partir d'un attribut de clone, mais à partir d'une expression.

Nous avons vu que pour un feature « f » possédant un attribut « a », « f.a » signifie une collection composée de valeurs des attributs « a » des clones de « f ».

Par exemple, considérons un feature X clonable sous un feature racine R. Le feature X est composé d'un sous-feature Y clonable. Soit une expression de la forme suivante :

```
forAll(X){  //corps de condition }
```

Dans le corps de la condition de l'expression ci-dessus, « Y.a » signifierait la collection des valeurs des attributs « a » issues des clones de Y à prendre en compte dans le contexte (sous le clone de Y courant).

Si les clones à prendre en compte sont {y1, y2, … yk}, alors f {E} signifie la collection {E(y1), E(y2), … E(yk)}, où E(yi) est la valeur de l'expression E évaluée dans le contexte de yi.

Cela permettrait de fournir à une fonction d'agrégation non pas une collection de valeurs d'attributs issues d'un set de clones mais une collection de valeurs calculées à partir d'autres fonctions d'agrégation portant sur les descendants de ses clones ainsi que d'autres clones « cousins ».

Reprenons le produit décrit par la figure 6.2 en guise d'illustration.

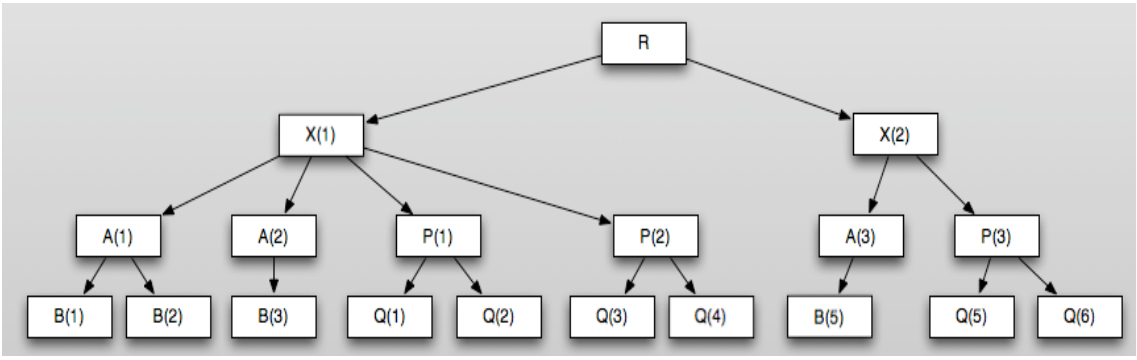


Figure 6.2

Soit la contrainte : *Pour tout clone de A, la somme des valeurs des attributs price de clones de B doit être inférieure aux sommes des attributs price des clones de Q calculées pour chaque clone de P ayant un ancêtre commun de type X avec le clone de A.*

Voici son expression dans le langage de contraintes :

```
forAll(X){ forAll(A){
           sum(B.price) < min( X.P {sum(Q.price) } )
     }}
```

Nous constatons que la somme des valeurs de « B.price » doit être comparée avec le minimum d'une collection de valeurs dont chaque valeur est calculée à partir d'un clone de P. L'utilisation d'un attribut simple ne permettrait pas d'exprimer cette contrainte, les valeurs calculées sous la forme {E} sont donc nécessaires dans ce cas.

La syntaxe d'une collection d'expressions calculées est similaire à celles des collections de valeurs d'attributs, si ce n'est la présence d'une expression en lieu et place de l'attribut :

```
COL_EXPR  ::= SET_CLONE {E}
```

Une telle collection d'expressions peut être utilisée en argument d'une fonction d'agrégation.

Comme décrit précédemment, cette expression va donc permettre de constituer un set de clones dont le type est celui du dernier feature cité dans le chaine. Pour chacun de ces clones du set, l'expression E sera évaluée en considérant que le clone courant est le clone évalué.

La sémantique est triviale, elle est similaire à la sémantique des attributs qualifiés, hormis le fait que les valeurs des attributs « a » sont remplacées par les valeurs des expressions « E ».

6.3.6 Count

Une fonction count n'était pas nécessaire dans l'ancienne version de la syntaxe abstraite.

En effet, le count était utilisé, dans la syntaxe concrète, pour compter le nombre de features présents dans le produit sous un feature parent. Or, un nom de feature présent dans une expression était évalué à true si le feature était sélectionné.

Comme décrit par « Syntax & Semantic of TVL » [20], une expression telle que (c1 ? 1 : 0) + … + (ck ? 1 : 0), où {c1 ,…ck} sont les sous-features possibles (définis dans le modèle) d'un feature p, permettait donc de compter le nombre de features présents dans le produit sous le feature p.

Mais suite à l'introduction du clonage, une telle expression ne convient plus. En effet, nous avons vu au point 6.3.1 qu'un nom de feature présent dans une expression n'est plus évalué à true si le feature est sélectionné, puisqu'un feature clonable peut être représenté par plusieurs instances.

Une fonction « count » est donc nécessaire, mais plusieurs solutions sont possibles :

- Compter le nombre d'instances d'un feature sous une instance donnée,
  - parmi les enfants du clone donné
  - parmi tous les descendants du clone donné
- Compter le nombre de clones présents dans un set de clones (cfr 6.3.3).

Par cohérence avec l'utilisation de set de clones (cfr 6.3.3 et 6.3.4) et la définition de l'opérateur ensembliste #, count permettra de comptabiliser le nombre d'éléments dans un set de clones, il n'est toutefois pas nécessaire de compter le nombre d'éléments dans une collection de valeurs, puisque ces valeurs sont obtenues à partir des membres d'un set de clones.

Sa syntaxe est la suivante : E ::= count(SET_CLONE)

Sa sémantique est similaire à celle de l'opérateur # :

Soit C un set de clones {c1, c2, … ck}, représenté par SET_CLONE
[[count(SET_CLONE)]] = # {c1, c2, … ck}

6.4 Transposition à TVL

Cette section vise à transposer les modifications apportées à la syntaxe abstraite (cfr 6.3) à la syntaxe concrète ainsi qu'à adapter les éléments de la syntaxe concrète n'appartenant pas à la syntaxe normalisée de TVL.

6.4.1 Récapitulatif des modifications de la syntaxe abstraire

Voici un récapitulatif de ces modifications de la syntaxe abstraite :

- Suppression de `B ::= f`
- Ajout de `B ::= forAll(f){B}` et `B ::= exists(f){B}`
- Suppression de « `exists` » et « `requires` »
- Construction de collections de valeurs d'attributs avec filtres.
- Construction de collections de valeurs à partir des expressions calculées avec filtres.

6.4.2 Adaptations des « sucres syntaxiques »

Avant d'intégrer les modifications ci-dessus à la syntaxe abstraite, considérons à présent les éléments de TVL n'appartenant pas à TVL_{NF} :

- **root, this, parent**

  Ces mots-clés étaient utilisés afin de permettre de référencer le feature racine, le feature courant et le feature parent d'un feature.

  Vu l'introduction de la notation « p.f.a », ces features ne sont plus nécessaires et pourraient prêter à confusion.

  Par exemple, considérons l'expression suivante :

```
forAll(X){ forAll(A){
      total > 100 &&
      sum(B.price) < min( X.P[code = 1] {sum(Q.price)} )
}}
```

  « total » représente un attribut de A et code un attribut de P.

  Si le mot clé « this » était utilisé afin de représenter le clone courant, l'expression deviendrait :

```
forAll(X){ forAll(A){
      this.total > 100 &&
      sum(B.price)< min( X.P[this.code = 1] {sum(Q.price)} )
}}
```

  Le mot-clé « this » aurait donc deux significations différentes dans la même expression, ce qui est assez perturbant. Il en irait de même pour le mot-clé « parent ».

  Etant des « sucres syntaxiques », les mots-clés « root », « this » et « parent » sont supprimés de la syntaxe concrète.

- **Children**

  Ce mot-clé de TVL permettait de sélectionner les sous-features d'un feature. Le set de sous-features ne dépendait pas des features sélectionnés dans le produit, mais uniquement des décompositions définies dans le modèle.

  Vu l'introduction du clonage, un produit n'est plus composé de features mais d'instances de ceux-ci.

  Sélectionner tous les sous-features définis dans le modèle permettait de manipuler une collection de features contenant tous les sous-features possibles d'un feature (un « produit complet »), mais dans le cadre d'un produit pouvant comporter dans certains cas un nombre illimité d'instances, le set composé de tous les sous-features possibles ne représente plus ce produit « complet ».

  De plus, ce mot-clé peut prêter à confusion puisqu'un clone n'est pas un feature.
  Children sera donc supprimé de la syntaxe concrète.

- **Selectedchildren**

  Contrairement à children, selectedChildren considérait uniquement les features sélectionnés dans le produit. Mais puisque le produit est constitué de clones, une adaptation de selectedChildren pourrait se faire de deux manières :
  - Soit selectedChildren représente les clones enfants du clone courant, ce qui est déjà permis avec une notation « p.f.a » avec valeurs calculées.
  - Soit selectedChildren représente les types de features représentés sous un clone. Cela a peu d'intérêt dans les cas industriels et peut prêter à confusion car manipule des features et non des clones.

  Le mot clé selectedChildren est donc supprimé de la syntaxe concrète, il sera remplacé par l'utilisation « p.f.a ».

- **Count**
  Count pouvait être utilisé sous ces formes : count(children) et count(selectedChildren), il permettait donc de compter le nombre de sous-features présents sous un feature (dans le modèle ou dans le produit).

  Children et selectedChildren étaient les seuls mots-clés permettant de représenter des collections d'éléments. Vu leur remplacement par la syntaxe « p.f.a », le count de TVL va être adapté afin de permettre de compter le nombre de clones présents dans un set de clones.

  Le count de TVL repose sur la définition sémantique du count définit dans le langage abstrait, permettant de compter le nombre de clones présents dans un set de clones.

6.4.3 Grammaire expressions de TVL

Après avoir déterminé les modifications de la syntaxe abstraite et les modifications à apporter à des éléments propres à la syntaxe concrète, nous pouvons à présent décrire la nouvelle version de la grammaire concrète des expressions de TVL :

L'ancienne version de cette grammaire est détaillée dans « Syntax & Semantic of TVL » [20]. Les adaptations apportées à cette grammaire sont **en gras.**

ID = ("a"-"z" | "A"-"Z") ("a"-"z" | "A"-"Z" | "0-9" | "_") * ;
(Ids composés, pas de root, parent ni this)
LONG_ID = ID | ID "." LONG_ID ;

EXPRESSION =
(*Référence à un attribut, qualifié ou non*)
LONG_ID

(*Quantificateurs forAll et exists*)
| forAll(ID){EXPRESSION}
| exists(ID){EXPRESSION}

(*Set de clones avec filtre*)
| SET_CLONE ::= (ID [«[» EXPRESSION «]»] « . »)* ID[«[» EXPRESSION «]»]

(*Attribut qualifié représentant une collection de valeurs d'attributs basée sur un set of clones*)
| QUALIFIED ::= SET_CLONE « . » ID

(* Collection de valeurs calculées à partir d'un set de clones *)
|COL_EXPR ::= SET_CLONE {EXPRESSION}
(* Groupe *)
| "(" EXPRESSION ")"

(*Expression conditionelle*)
| EXPRESSION "?" EXPRESSION ":" EXPRESSION

(* Expression booléenne *)
| EXPRESSION "&&" EXPRESSION
| EXPRESSION "||" EXPRESSION
| EXPRESSION "->" EXPRESSION (* implication *)
| EXPRESSION "<-" EXPRESSION
| EXPRESSION "<->" EXPRESSION (* equivalence *)
| "!" EXPRESSION
| "true"
| "false"

(*Agrégation booléenne*)
| "and" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "or" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "xor" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"

(* Comparaison *)
| EXPRESSION "==" EXPRESSION
| EXPRESSION "!=" EXPRESSION
| EXPRESSION "<=" EXPRESSION
| EXPRESSION "<" EXPRESSION
| EXPRESSION ">=" EXPRESSION
| EXPRESSION ">" EXPRESSION

(*Restriction du domaine*)
| EXPRESSION "in" SET_EXPRESSION

(* Arithmétique *)
| EXPRESSION "+" EXPRESSION
| EXPRESSION "-" EXPRESSION

```
| EXPRESSION "/" EXPRESSION
| EXPRESSION "*" EXPRESSION
| "-" EXPRESSION
| "abs" "(" EXPRESSION ")"
| INTEGER | REAL

(*Agrégation arithmetique *)
| "sum" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "mul" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "min" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "max" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "avg" "(" (EXPRESSION_LIST | CHILDREN_ID) ")" ;

(*count utilisant un set de clones*)
| "count" "(" SET_CLONE ")"
```

6.4.4 Contraintes sémantiques

Comme abordé au point 3.5, la grammaire ne permet pas de garantir certaines contraintes sémantiques telles que la portée des identificateurs et le typage. Les modifications apportées à la grammaire des expressions de TVL sont concernées par ce problème.

Ce point ne détaille pas toutes les contraintes à implémenter par l'analyseur syntaxique puisque ce chapitre n'est qu'une présentation des modifications à apporter au langage des expressions et non la réalisation d'un parser le mettant en œuvre. En outre, ces contraintes ont été définies lors de la description des modifications de la syntaxe abstraite et de sa sémantique.

Les exemples suivants visent donc à illustrer les différents types de problèmes à résoudre par l'analyseur sémantique à l'aide de quelques contraintes issues de la description du nouveau langage d'expressions.

Nous avons vu dans la présentation des modifications de la syntaxe abstraite et de la syntaxe concrète que les IDs représentent des features ou des attributs. Ceux-ci doivent bien entendu exister dans le modèle et respecter la hiérarchie de features du modèle ainsi que le contexte d'utilisation.

Par exemple, un attribut qualifié est constitué d'une suite de noms de features où le dernier est le feature propriétaire de l'attribut. Cet attribut doit donc exister et appartenir à ce feature. Les features précédents dans la suite doivent être des ancêtres du feature propriétaire et ancêtres l'un de l'autre. Ces contraintes doivent être vérifiées par l'analyseur sémantique.

Un autre exemple, un attribut qualifié peut représenter une collection de valeurs d'attributs (cfr 6.3.3). Si c'est le cas, il ne peut pas être utilisé dans une comparaison avec une valeur scalaire, alors qu'il le peut s'il est associé à une seule valeur. L'analyseur sémantique devra donc vérifier que le feature propriétaire et aucun de ses parents ne puissent être clonés, afin de garantir que le feature propriétaire de l'attribut ne puisse engendrer qu'une seule valeur. Une autre solution serait de ne pas vérifier cette contrainte de manière statique mais d'attendre l'instanciation d'un produit et d'y vérifier qu'une seule valeur existe.

Enfin, le typage est aussi à vérifier. Par exemple, les expressions utilisées comme conditions ou dans le corps des quantificateurs doivent être booléennes, même si elles peuvent se composer d'expressions arithmétiques.

6.4.5 Normalisation

Comme défini au point 6.1, la normalisation permet d'exprimer certaines constructions considérées comme des « sucres syntaxiques » de la syntaxe concrète des expressions de TVL dans des expressions sémantiquement équivalentes mais n'utilisant qu'un sous-ensemble de la syntaxe concrète de TVL

Nous avons vu au point 6.4.2 que les sucres syntaxiques sont soit supprimés soit exprimables avec un sous-ensemble de TVL possédant une équivalence dans le langage abstrait. Il n'y a donc pas de modification de la normalisation concernant ces constructions.

Une des étapes de la normalisation, décrite dans « Syntax & Semantic of TVL » [20], consiste à attacher les contraintes exprimées sur certains features à la racine du modèle. En effet, dans la version normalisée, toutes les contraintes sont attachées à la racine et non à des features en particulier.

Or, nous avons vu dans les différents éléments présentés dans la section 6.3, que l'évaluation des expressions dépend toujours du clone courant. Déplacer l'expression d'une contrainte d'un feature vers la racine influence-t-il l'interprétation de cette expression ?

Non, puisque les quantificateurs s'enchainent à partir de la racine et s'il n'y a pas de quantificateur, le clone courant est la racine. A l'intérieur du corps du quantificateur, on s'appuie sur le clone courant défini par le quantificateur, il n'y a donc pas de différence que la contrainte soit déclarée dans le corps de la racine ou dans le corps d'un autre feature.

6.5 Limites et travaux futurs.

La solution proposée répond à l'ambiguïté causée par l'introduction du clonage à l'aide de quantificateurs basés sur les quantificateurs mathématiques ∀ et ∃, dont le langage OCL s'est aussi inspiré.

A ceux-ci s'ajoute des filtres et la définition d'expressions calculées permettant de sélectionner des ensembles de clones répondant à une condition bien précise et de calculer des valeurs à partir des valeurs d'attributs de clones et de formules de calculs.

Les choix effectués reposent sur une base mathématique et s'inspire de fonctionnalités offertes par OCL afin de couvrir le plus grand nombre de cas possibles tout en évitant une complexité trop importante.

Nous avons constaté dans les divers exemples que ces choix répondaient aux attentes, mais le nombre d'exemples étant infini, une confrontation des ces choix à un grand nombre de cas industriels, pouvant être plus complexes et plus volumineux, permettra d'évaluer et d'adapter les choix effectués en matière d'expressivité et de complexité.

Un autre futur travail consiste à implémenter cette nouvelle version de la syntaxe et de la sémantique des expressions des contraintes « cross-tree ». Cette implémentation doit être faite au niveau de l'analyseur syntaxique et sémantique de TVL afin de vérifier que les contraintes exprimées dans un fichier TVL sont correctes syntaxiquement et sémantiquement.

Suite à l'implémentation de ces contraintes, l'outil de vérification de la satisfaisabilité des modèles devra également est adapté si nécessaire afin de vérifier ces contraintes.

7. Langage de configuration : TVL-P

Les chapitres 2 et 4 ont respectivement introduit les feature diagrams et le langage TVL, les notions de modèles et de produits y ont été abordées. Toutefois, le langage TVL permet uniquement de définir des modèles. Ce chapitre vise donc à proposer une extension de TVL, destinée à décrire les produits.

Les motivations d'un tel langage sont présentées. Nous verrons ensuite qu'il existe un langage similaire développé dans le cadre du projet HATS, mais que dans le cadre de TVL, il est préférable de développer un nouveau langage (TVL-P) plutôt que d'utiliser le langage issu de HATS.

Ensuite, ce chapitre décrit la syntaxe, la grammaire et la sémantique de TVL-P.

7.1 Motivations

Les avantages d'une version textuelle de configuration du produit par rapport à l'utilisation d'une version graphique sont identiques à ceux du langage TVL par rapport aux feature diagrams (cfr chapitre 4). Pour rappel, une version textuelle est plus compacte qu'une version graphique, elle peut être traitée à l'aide d'outils d'édition et de parsing permettant la validation.

De plus, par cohérence avec l'expression des modèles en TVL, il est naturel d'exprimer les produits de façon analogue à l'expression des modèles, ce qui incite à dériver un langage de configuration à partir de TVL.

Dans la version antérieure à ce mémoire du parser TVL, les configurations des produits étaient exprimées sous forme de clauses.

Pour rappel, le mémoire P. Faber [24] présente l'utilisation d'un solver SAT [24,25] afin d'étudier la satisfaisabilité d'un modèle. Le modèle TVL était traduit en un problème SAT, composé d'un ensemble de clauses logiques, telles que : $(a1 \vee a2) \wedge (a2 \vee \neg a3) \wedge \neg a4$

Le moteur d'inférence du solver, comparable à celui de Prolog, pouvait ensuite être interrogé en lui spécifiant des valeurs booléennes de variables, par exemple : $a1 = false \wedge a4 = false$, afin d'exprimer la configuration du produit.

Une configuration d'un produit était donc dans ce cas un ensemble de conditions booléennes associées à un problème SAT, ce qui n'est pas très « user friendly » pour un utilisateur de TVL.

Une version textuelle inspirée de TVL est donc plus compréhensible et utilisable par un utilisateur de TVL. Cette configuration pourra également être éditée et validée à l'aide d'outils d'édition et de parsing (cfr chapitre 8) mais aussi sauvegardée, imprimée et transmise comme tout fichier texte.

Bien entendu, l'utilisation d'un solver capable de valider les contraintes « cross-tree » serait toujours nécessaire. Le langage TVL-P permettrait à l'utilisateur d'exprimer des configurations de manière intuitive, mais le solver pourrait ensuite être utilisé afin de vérifier que les configurations exprimées en TVL-P respectent bien les contraintes « cross-tree ».

7.2 Projet HATS

7.2.1 Présentation

Cette section, inspirée de Deliverable D1.2 [32] et de Clarke [33] présente brièvement le projet HATS.

Dans le cadre de ce projet, une framework permettant la modélisation et la génération de produits appartenant à des SPLs a été développée. Cette framework appelée « Full ABS Modelling Framework » utilise le langage « Full ABS Language » qui se compose du langage Core ABS et de ses 4 extensions, parmi lesquelles un langage de définition de feature models nommé µTVL (basé sur TVL) et un langage de sélection de features nommé PSL.

Afin de comprendre l'utilité des ces différents langages, en particulier µTVL et PSL, voici une description de la framework Full ABS, suivie d'un exemple.

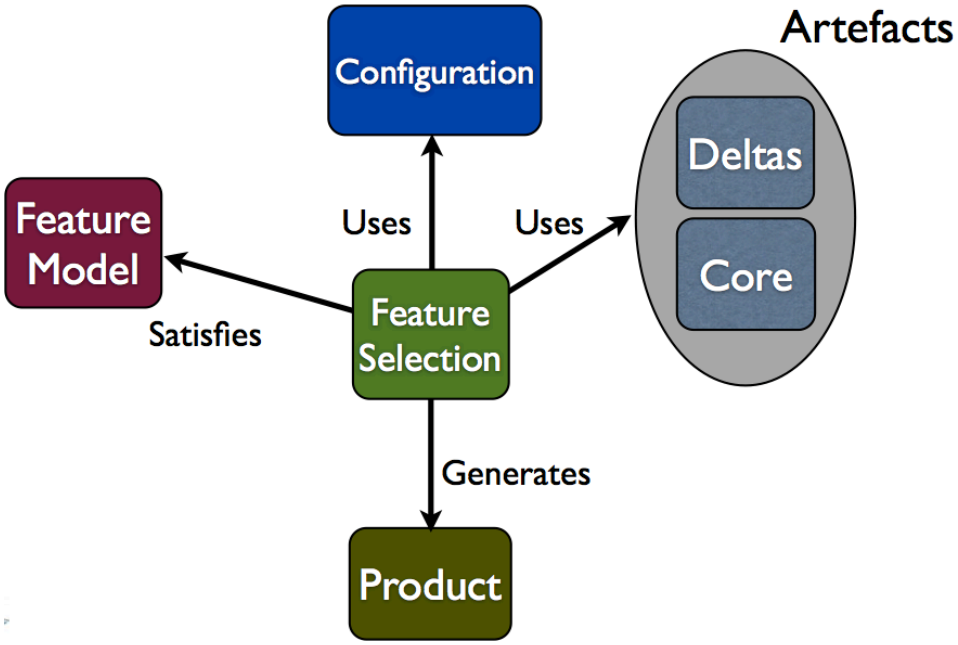


Figure 7.1 : Composants de Full ABS [33].

Le but de cette framework est de modéliser la variabilité des SPLs, mais aussi de générer des logiciels faisant partie d'une SPL.

Le feature model est comparable à un modèle exprimé en TVL, il est d'ailleurs exprimé en **µTVL**, une adaptation de TVL.

Les artefacts contiennent des éléments visant à définir le comportement du logiciel à générer. Le core défini le comportement de base, c'est une application à part entière, définie dans le langage **ABS Core**.

Les deltas sont les définitions d'altérations sur le comportement défini par le core. Ils permettent de créer/modifier/supprimer des classes et des méthodes de l'application. Le langage **DML** permet d'exprimer ces deltas.

La configuration associe des deltas aux features et défini dans quelles conditions et dans quel ordre des deltas doivent être appliqués au comportement défini par le core. Ces conditions d'applications sont principalement liées à la présence ou non de certains features dans le produit.

La sélection de feature consiste à spécifier quels features sont sélectionnés et quelles valeurs sont attribuées à leurs attributs. Cela permet donc d'appliquer certains deltas dans un certain ordre selon les features sélectionnés et les règles définies dans la configuration, le produit peut donc être généré en conséquence. Le langage **PSL** permet de définir les features sélectionnés et les valeurs attribuées à leurs attributs.

Les 5 langages suivants sont donc utilisés :

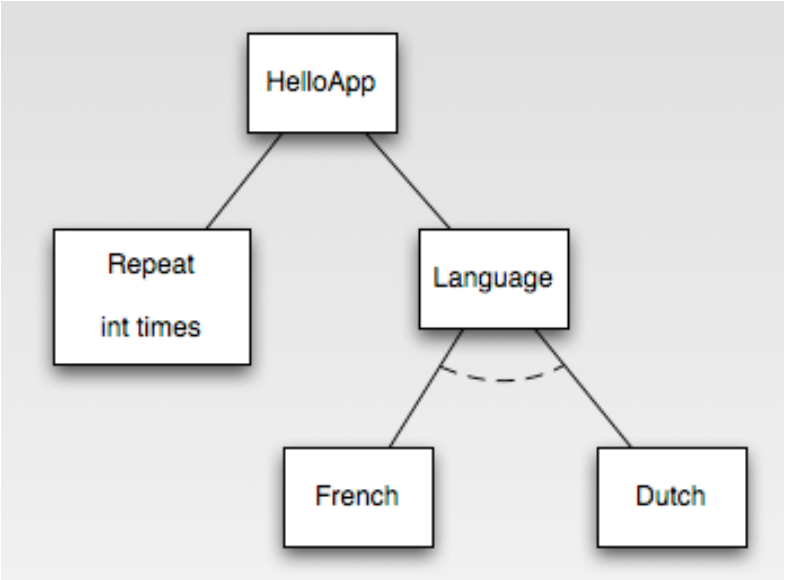
- **Core ABS** : modélisation des comportements.
- **µTVL** : description du feature model.
- **Delta Modelling Language (DML) :** définition des différences de comportement par rapport au core.
- **Configuration language (CL)** : règles d'applications des deltas selon les features sélectionnés et leurs attributs.
- **Product Selection Language (PSL)** : sélection des features et assignation de valeurs aux attributs.

Les points suivants vont décrire ces concepts avec comme fil conducteur un exemple inspiré d'exemples de [32] et [33].

7.2.2 Description du feature model

Considérons une SPL permettant de générer des applications dont le but est de dire bonjour un certain nombre de fois, en français ou en néerlandais, selon la configuration du produit généré.

Voici le feature diagram :



Voici la description en **µTVL** :

```
root HelloApp {
      group allof {
            Language {
                  Group oneof {French, Dutch}
            },
            Repeat {
                  Int times;
            }
      }
}
```

Ce langage est une adaptation allégée de TVL, sa grammaire est définie dans Deliverable D1.2 [32], en voici un extrait (les contraintes ne sont pas représentées dans cet extrait) :

Model	::=	(root *FeatureDecl*)* *FeatureExtension**
FeatureDecl	::=	FID [{ [*Group*] *AttributeDecl** *Constraint** }]
FeatureExtension	::=	extension FID { *AttributeDecl** *Constraint** }
Group	::=	group *Cardinality* { [opt] *FeatureDecl*, ([opt] *FeatureDecl*)* }
Cardinality	::=	allof | oneof | [n_1 .. *] | [n_1 .. n_2]
AttributeDecl	::=	Int AID ; | Int AID in [*Limit* .. *Limit*] ; | Bool AID ;
Limit	::=	n | *

Voici les principales différences par rapport à TVL (avec clonage) :

- Pas de cardinalité de feature
- Pas de feature « shared »
- Pas d'attribut composé, ni d'initialisation

En outre, la syntaxe diffère quelque peu à plusieurs niveaux (liste non-exhaustive):

- définition en extension : utilisation du mot-clé « extension ».
- définition des décompositions : le group doit être inclus dans la définition du corps, il n'existe pas de « sucre syntaxique » permettant de définir la décomposition à la suite du nom du feature.
- L'expression des domaines de valeurs est différente

7.2.3 Description des artefacts

La définition du core représente le comportement de base de l'application. Ce comportement est exprimé dans le langage Core ABS et est exécutable.

Voici l'interface et la classe permettant de retourner le message « bonjour » :

```
interface Greeting {
      String sayHello() ;
}

class Greeter implements Greeting {
      String sayHello () { return « Bonjour» ;}
}
```

Voici à présent à présent un delta défini dans le langage DML :

```
delta Nl {
      modifies class Greeter {
            modifies String sayHello(){
                  return « Goiedag» ;
            }
      }
}
```

Ce delta permet de modifier le comportement défini par le core pour la fonction sayHello de la classe Greeter.

Nous passons les détails de la classe principale de l'application HelloApp. Cette classe a pour but d'appeler la méthode « sayHello » un certain nombre de fois. La valeur de ce nombre n'est pas définie dans le core, mais un delta sera appliqué et définira cette valeur selon la valeur de l'attribut « times » du feature « Repeat ».

7.2.4 Configuration et sélection de features

La configuration, exprimée en langage CL, permet de définir les règles d'application des deltas ainsi que l'ordre d'application, en fonction des features sélectionnés et de leurs valeurs d'attributs. Voici la configuration de HelloApp :

```
Productline HelloApp{
      features Repeat, French, Dutch ;
      delta Rpt(Repeat.times) after Nl ;
      delta Nl when Dutch ;
}
```

La clause « `features …`» indique les features intervenant dans les règles.

La clause « `delta Nl when Dutch`» indique que si le feature « Dutch » est présent, le delta « Nl » est appliqué afin de modifier le message.

La clause « `delta Nl when Dutch`» signifie que le delta « Rpt » sera toujours appliqué (pas de « `when` ») afin de configurer le nombre de répétitions du message selon « Repeat.times ». Ce delta sera moins prioritaire que « Nl » (`after Nl`).
Nous avons vu la description du feature model, des artefacts et de la configuration. La sélection des features va donc avoir un rôle central, elle permettra de définir quels features sont présents et quelles sont les valeurs des attributs. La framework pourra, selon cette sélection et la configuration, déterminer quels deltas appliquer au core, le comportement de l'application pourra donc être défini automatiquement par la framework.

Voici une sélection parmi les features de HelloApp :

```
Product P1 (Dutch, Repeat {times = 3}){
      New Application() ;
}
```

Cette configuration est exprimée en PSL, en voici la grammaire (issue de Deliverable D1.2 [32]) :

Selection ::= `product` *TypeId* (*FeatureSpecs*) { *InitBlock* }

FeatureSpecs ::= *FeatureSpec* (, *FeatureSpec*)*
FeatureSpec ::= FID [*AttributeAssignments*]

AttributeAssignments ::= { *AttributeAssignment* (, *AttributeAssignment*)* }
AttributeAssignment ::= AID = *Literal*

InitBlock ::= *Block*

Il s'agit donc du nom du produit (P1) suivi d'une suite de features sélectionnés pour lesquels l'on peut assigner les valeurs des attributs.

Aucune notion de hiérarchie n'est présente dans cette grammaire. De plus, les concepts absents de µTVL par rapport à TVL sont aussi absents de PSL (cardinalités de features, types composés).

7.2.5 Conclusion : PSL ou TVL-P ?

Une des principales motivations de l'utilisation d'un langage textuel de configuration des produits est de pouvoir décrire les produits de façon cohérente avec la description dans modèles TVL, et ce de manière intuitive pour l'utilisateur, tant lors de la définition que lors de la lecture de la configuration.

Dans cette optique, PSL ne convient pas suffisamment pour diverses raisons :

- Il est dérivé d'un langage inspiré d'une ancienne version de TVL, le clonage y est absent et d'autres concepts ont été modifiés ou supprimés.

- Les liens hiérarchiques ne sont pas représentés, ce qui va à l'encontre du souhait de cohérence au niveau ergonomique entre l'expression du modèle et du produit, ainsi que l'intuitivité souhaitée au niveau tant du modèle que du produit. En PSL, la lecture de la configuration du produit n'a pas de sens sans la lecture du modèle puisque les liens hiérarchiques sont absents.

Donc, plutôt que de mener une réflexion afin de modifier un langage (PSL) dérivé d'un langage (µTVL) étant lui-même une adaptation de l'ancienne version de TVL, il est préférable de mener cette réflexion directement à partir de la nouvelle version de TVL. TVL-P est le langage issu de cette réflexion. Les points suivants détaillent les différentes étapes de cette réflexion, allant de la spécification des exigences jusqu'à son implémentation.

7.3 Exigences

Le langage TVL-P est une extension du langage TVL destiné à représenter des produits issus de modèle TVL et utilisés par les mêmes utilisateurs que le langage TVL.

Les deux principes de bases à retenir lors de l'élaboration de ce langage sont :

- La cohérence avec TVL, puisque TVL-P en est une extension et est destiné aux mêmes utilisateurs.
- La simplicité de la syntaxe, car le langage n'est pas destiné à être manipulé uniquement par des machines mais aussi par des humains.

Afin de représenter les produits en respectant ces deux contraintes, les éléments de la syntaxe de TVL seront transposés à la syntaxe de TVL-P s'ils sont nécessaires à l'expression d'un produit.

Comme vu dans le chapitre 4, TVL se compose des éléments suivants :

- Hiérarchie de features
- Attributs
- Contraintes

Les exigences suivantes en sont donc déduites :

- TVL-P devra représenter quels features ont été sélectionnés dans le produit, quels sont leurs clones et quelles sont les relations de décomposition entre les clones.

- Un produit peut attribuer des valeurs aux attributs décrits dans le modèle. TVL-P devra donc permettre l'attribution de ces valeurs.

- Par contre, nous faisons l'hypothèse pour cette première version que les contraintes appartiennent au modèle, un produit valide doit vérifier ces contraintes, mais ne peut pas en définir de nouvelle. TVL-P ne devra donc pas être en mesure d'exprimer des contraintes.

En outre, le langage TVL-P utilisera les mêmes caractères que TVL, permettra l'utilisation de commentaires ainsi que des inclusions de fichiers de façon similaire à TVL.

7.4 Syntaxe

7.4.1 Représentation des clones et de leur hiérarchie

Voici un exemple de Modèle TVL:

```
root Document{

    group allOf{

        Type group oneOf{
            Normal,
            Booklet
        },
        Sheet [1..*] group oneOf{
            Folding,
            Page group allOf{
                Orient group oneOf{
                    Portrait,
                    Landscape
                }
            }
        }

    }
}
```

Dans ce modèle, le feature « Document » se décompose en un feature « Type » ET un ou plusieurs features « Sheet ».

« Type » et « Sheet » sont décomposés dans une décomposition de type « oneOf », ce qui signifie les produits doivent comporter au moins un type de feature parmi les enfants de « Type » et « Sheet ».

Figure 7.2 – Modèle TVL

Voici un exemple de produit issu de ce modèle :

```
Document{

    group{

        Type group{
            Normal
        },

        Sheet as Sheet1 group {
            Folding,
            Page group {
                Orient group {
                    Portrait
                }
            }
        },

        Sheet as Sheet2  group {
            Page group {
                Orient group {
                    Landscape
                }
            }
        }

    }
}
```

Figure 7.3 Produit issu du modèle 7.2

Root

Nous constatons que le mot-clé « root » n'est pas présent dans le produit. La racine est définie dans le modèle et doit être unique, utiliser le mot-clé « root » dans le produit serait donc redondant puisque seul l'unique instance du feature « Document » pourrait être précédée de « root », ce qui est déjà défini dans le modèle.
La seule utilité serait donc de rappeler aux lecteurs du produit quel est l'élément racine. Afin de simplifier les traitements dans cette première version du langage, le mot-clé « root » n'est pas permis dans le produit.

Ce produit « Document » est composé d'une instance du feature « Type » et de deux clones du feature « Sheet ».

L'instance du feature « Type » est composée d'un feature « Normal », il s'agissait dans le modèle d'une décomposition de type « oneOf », il est donc correct qu'un seul des sous-features soit sélectionné.

Group

De façon similaire à la définition du modèle, le mot-clé « group » est utilisé pour définir les décompositions de « Document » et « Type », dans lesquelles sont listés les instances des features sélectionnés.

Ce mot-clé « group » peut être utilisé dans le corps de la définition du contenu d'une instance de feature, ce qui permet, si nécessaire, de placer des assignations de valeurs aux attributs (cfr 7.4.3) dans le corps de l'instance du feature :

```
Document {
      //valeurs des attributs
      …
      //décomposition
      group {…}
}
```

Mais « group » peut aussi être placé directement à la suite du nom du feature à condition que la décomposition soit la seule information à fournir dans le corps de cette instance, par exemple : `Type group {Normal}`

Une définition sans le mot « group » a été envisagée, mais la présence seule des {} combinée à la présence d'assignations d'attributs serait moins lisible et ne serait pas cohérente avec TVL, exemple :

```
Document {
      //valeurs des attributs
      …
      //décomposition
      {…}
}
```

A la suite de l'instance de « Type », deux clones de « Sheet » sont définis. Leur structure est également définie avec le mot-clé « group », et ainsi de suite avec les instances de leurs sous-features.

L'élément « as … » sera défini dans le point 7.4.4.

Cardinalités de groupe

Pour chaque décomposition, aucune cardinalité n'est présente, que ce soit sous la forme [x..y] ou des mots-clés « oneOf », « allOff », « someOf » .

En effet, la définition des cardinalités de décomposition incombe au modèle, le produit devra alors respecter ces cardinalités.

Cardinalités de feature

Il en va de même pour les cardinalités de feature, elles définissent le nombre de clones autorisés pour un feature donné, elles n'ont donc aucun sens dans le produit, mais ce produit doit les respecter. C'est d'ailleurs le cas ici, la cardinalité du feature « Sheet » est [1..*] , or il y a deux clones de « Sheet », ce qui est correcte.
Le mot-clé « opt » ayant la même signification qu'une cardinalité de feature [0..1], il est aussi interdit dans TVL-P.

7.4.2 Cas particulier « Shared »

Voici un exemple de modèle comportant des features « shared ».

```
root House{
    group allOf{
        Kitchen group allOf {Door},
        Living group allOf {shared Door}
    }
}
```

Ce modèle « House » se compose de deux sous-features, « Kitchen » et « Living », tout deux partageant le feature « Door ».

Voici un produit issu de ce modèle :

```
House group{
    Kitchen group {Door},
    Living group {Door}
}
```

Ce produit se compose d'une instance de « Kitchen » et d'une instance de « Living » partageant la même instance de « Door ».

L'utilisation de « shared » dans le produit n'est pas nécessaire, puisque TVL impose que dans le cas d'un feature partagé, aucun autre feature différent de ce feature partagé ne porte le même nom.

Cela implique que tout clone de « Door » dans le produit ne pourra être qu'un clone du feature « Door » partagé.

Aucune notation particulière n'est donc nécessaire. Le caractère partagé du feature est indiqué dans le modèle.

Toutefois, l'indiquer dans le produit permettrait de rappeler aux lecteurs que ce feature est un feature partagé sans qu'il ne doive consulter le modèle, mais cela impliquerait des

vérifications sémantiques afin de vérifier que le mot-clé « shared » est placé dans un produit sur des clones qui instancient un feature « shared » du modèle.

Afin d'éviter de complexifier ces traitements, le mot-clé « shared» n'est pas utilisé dans le produit.

Remarque : L'utilisateur peut évidemment ajouter un commentaire dans le produit afin d'indiquer qu'il s'agit d'un clone d'un feature partagé. Les commentaires sont libres et ne sont soumis à aucune vérification sémantique.

7.4.3 Attributs

TVL permet d'assigner des valeurs aux attributs, qu'ils soient simples ou composés.

Par cohérence, TVL-P se basera donc sur la syntaxe de TVL, mais en réduisant celle-ci. En effet, la définition de contraintes portant sur les valeurs des attributs est de la responsabilité de modèle et non du produit, ce qui implique que :

- Les expressions utilisées seront des valeurs et non des expressions exprimant des valeurs calculées à partir d'autres attributs.
- Les restrictions de domaines de valeurs seront également interdites dans TVL-P.

En outre, les assignations conditionnelles utilisant « ifIn » et « ifOut » seront également interdites dans TVL-P, puisqu'elles n'ont aucune utilité dans le produit. En effet, si un feature est présent dans la description du produit, c'est qu'il a été sélectionné dans ce produit, utiliser « ifIn » et « ifOut » pour déterminer des valeurs différentes selon que le feature soit sélectionné ou pas n'a donc pas de sens.

La définition de types, simples ou composés, est également réservée au modèle. Ces types ne seront d'ailleurs pas rappelés dans le produit puisqu'ils n'auraient qu'un rôle indicatif et leur présence impliquerait des traitements de vérifications sémantiques afin de vérifier que les types cités dans le produit correspondent bien aux types déclarés dans le modèle.

La définition de constantes n'est pas permise dans cette première version de TVL-P. Une constante définie dans le modèle peut être utilisée dans tous les produits issus de ce modèle, mais une constante définie dans un produit ne pourrait être utilisée que dans ce produit. Vu cette portée limitée, la définition de constantes locales au produit a donc peu d'intérêt.

Voici un exemple de modèle comportant des attributs :

```
struct record
    {
        int i;
        real r;
        enum orientation in {H, V, X};
        bool b;
    }

root Test {
    int a;
    int b;
    record myRecord;
}
```

Cet exemple contient des attributs simples « a » et « b », ainsi qu'un attribut composé « myRecord » dont la structure est définie par le type « record ».

Voici un exemple d'assignations de valeurs à ces attributs dans un produit :

```
Test {

    a is 1;

    myRecord
    {
        i is 1;
        orientation is H;
        b is true;
    }
}
```

La valeur d'un attribut simple est assignée avec l'opérateur « is » suivi d'une expression définissant sa valeur.

L'expression définissant la valeur d'un attribut dans TVL peut contenir des opérateurs arithmétiques ainsi que des références à des valeurs d'autres attributs. Les expressions dans cette première version de TVL-P seront plus limitées, elles seront uniquement composées de valeurs telles que : nombre entier, nombre réel, booléen, chaine de caractère ou valeur d'énumération.

La valeur d'un attribut composé est assignée en utilisant un bloc { } qui contient des assignations de valeurs pour les composants de l'attribut composé.

A l'intérieur d'un bloc, chaque assignation doit se faire en utilisant le nom du sous-attribut suivi de « is ». Une assignation de manière « anonyme » telle que {1 ; H ; true} n'est pas permise. Bien que cette écriture serait plus concise, elle impliquerait une notion d'ordre et de complétude dans la déclaration des attributs.

Assigner une valeur à tous les attributs ou sous-attributs n'est pas obligatoire, un attribut absent du produit sera considéré comme non initialisé. C'est le cas dans cet exemple de l'attribut « b » du feature « Test » et du sous-attribut « r » du type « record ».

7.4.4 Extensions

Comme décrit au point 4.2.1, TVL permet la définition du contenu d'un feature en extension. Ce mécanisme offre plus de lisibilité aux modèles et peut également être utile au produit pour la même raison.

Toutefois, la présence de plusieurs clones d'une même feature pose problème, reprenons l'exemple du produit de la figure 7.2 contenant un feature racine « Document » composé d'un feature « Type» et d'un feature clonable « Sheet » .

```
root Document{

    group allOf{

        Type group oneOf{
            Normal,
            Booklet
        },
        Sheet [1..*] group oneOf{
            Folding,
            Page group allOf{
                Orient group oneOf{
                    Portrait,
                    Landscape
                }
            }
        }

    }
}
```

Imaginons le cas d'un produit issu de ce modèle où plusieurs clones de « Sheet » seraient présents:

```
Document{

    group{

        Type group{
            Normal
        },

        Sheet,
        Sheet
    }
}
```

Si un seul clone de « Sheet » était présent, une définition en extension de « Sheet » respectant la même syntaxe que les extensions dans TVL pourrait être la suivante :

```
Document.Sheet group {
    Folding,
    Page group {
        Orient group {
            Portrait
        }
    }
}
```

Un identifiant long est utilisé afin d'indiquer de quel feature il s'agit. Mais dans le cas de clones multiples, comment déterminer sur quel clone porte l'extension ? Les « alias » vont être utilisés, le produit devient donc comme ceci :

```
Document{

    group{

        Type group{
            Normal
        },

        Sheet as Sheet1,
        Sheet as Sheet2
    }
}
```

La présence des alias « Sheet1 » et « Sheet2 » permet de référencer explicitement les clones afin d'y ajouter des définitions en extension. Celles-ci pourraient alors être comme ceci :

```
Document.Sheet1 group {
    Folding,
    Page group {
        Orient group {
            Portrait
        }
    }
}

Document.Sheet2  group {
    Page group {
        Orient group {
            Landscape
        }
    }
}
```

L'identifiant est constitué dans ce cas de l'alias donné au clone et non du nom du feature pour lequel différents clones sont présents. Bien entendu, dans le cas d'une instance unique d'un feature, le nom du feature peut être utilisé.

7.5 Définition complète ou partielle

Les éléments de la syntaxe peuvent se résumer en deux types :

- Définition de la hiérarchie
- Assignations de valeurs aux attributs

Un produit sera défini partiellement quand sa définition laissera une part de variabilité. Cela sera le cas quand certains attributs définis dans le modèle n'auront pas reçu de valeurs (ni dans le modèle ni dans le produit) ou quand la définition de la hiérarchie ne sera pas complète.

La définition de la hiérarchie est complète quand la décomposition de la racine est définie, que tous les sous-features soient sélectionnés ou non, et que la décomposition de chaque sous-feature est définie, et ainsi de suite récursivement avec les enfants des sous-features sélectionnés. Si la décompostion d'un sous-feature sélectionné n'est pas définie, la définition est partielle.

7.6 Grammaire

Voici à présent la grammaire au format EBNF du langage TVL-P. Celle-ci intègre les différents éléments présentés au point 7.4.

La définition d'un produit est constituée d'une collection d'éléments « FEATURE » représentant une instance du feature racine et sa descendance ainsi que des définitions en extension.

```
----- Starting point ------------------------------------------------------------
PRODUCT             = ( FEATURE )*

----- ID section ----------------------------------------------------------------
LONG_ID                 =  ID
                         | ID "." LONG_ID
```

Un élément « FEATURE » permet de définir le corps d'un clone. Ce corps est composé d'une liste FEATURE_BODY_ITEM pouvant être composée d'une seule décomposition (FEATURE_GROUP), dont l'unicité ne pourra être garantie par cette grammaire mais devra l'être par un analyseur sémantique, et d'assignations de valeurs d'attributs (ATTRIBUTE).

```
----- Feature section -----------------------------------------------------------
FEATURE         =  LONG_ID ("as" ID)? "{" FEATURE_BODY_ITEM* "}"
                 | LONG_ID ("as" ID)? FEATURE_GROUP

FEATURE_BODY_ITEM       =  data
                         | ATTRIBUTE
                         | FEATURE_GROUP

FEATURE_GROUP           = "group" "{" HIERARCHICAL_FEATURE ("," HIERARCHICAL_FEATURE)* "}"

HIERARCHICAL_FEATURE    =  FEATURE
                         | LONG_ID ("as" ID)?
```

Dans la décomposition d'un clone, certains clones peuvent être cités sans déclaration de leur corps. Ils pourront alors être définis en extension ou pas définis dans le produit (définition partielle). Ceci est permis par l'utilisation de LONG_ID dans la règle HIERARCHICAL_FEATURE.

Les attributs sont soit simples soit composés. L'assignation est toujours composée du nom de l'attribut suivie soit de l'affectation d'une expression avec l'opérateur « is » soit d'un bloc composé d'assignations de sous-attributs simples.

```
----- Attribute section -----------------------------------
ATTRIBUTE               =  ID ATTRIBUTE_BODY

ATTRIBUTE_BODY          =  "is" EXPRESSION ";"
                         | "{" (ID "is" EXPRESSION ";")* "}"
```

Une expression est un nombre entier, un nombre réel, une chaine de caractères, une valeur booléenne ou un ID d'une constante définie dans le modèle.

```
----- Expression section -----------
EXPRESSION              =  "true"
                         | "false"
                         | INTEGER
                         | REAL
                         | STRING
                         | ID
```

L'encodage de données libres est également permis sous le même format que dans TVL. Ces données peuvent par exemple contenir des informations associées aux fichiers TVLP et destinées à des outils manipulant le fichier tels que des éditeurs graphiques, parseur ou tout autre outil imaginable, puisque ces données sont libres.

```
----- Data section ----------------------------------
DATA                    = "data" "{" DATA_PAIR+ "}"
DATA_PAIR               = STRING STRING ";"
```

Enfin voici la description des littéraux utiliser dans le fichier TVLP. Ils sont similaires à ceux rencontrés dans TVL.

```
----- Value section ------------------------------------------------------
NATURAL                  = "0" | ["1"-"9"]["0"-"9"]*
INTEGER                  = "0" | ("-")?["1"-"9"]["0"-"9"]*
REAL                     = INTEGER "." (["0"-"9"]*["1"-"9"])?
ID                       = ["a"-"z" "A"-"Z"] ["a"-"z" "A"-"Z" "0" -"9" "_"]+
STRING                   = " " " [^*] " " "
```

7.7 Règles non vérifiées par la grammaire.

Comme c'est le cas de TVL, la grammaire de TVLP ne peut pas vérifier certaines règles :

- Un clone ne peut avoir qu'une seule définition de décomposition.
- Un alias doit être unique.
- Un attribut ne peut se voir attribuer qu'une seule valeur.

Et d'autres règles sont issues du modèle et devront être validées par rapport à lui :

- Les noms de features doivent exister dans le modèle et respecter la structure hiérarchique du modèle.
- Les noms d'attributs et leur valeur doivent respecter les attributs définis dans le modèle et leurs contraintes.
- Le produit doit respecter l'ensemble des contraintes exprimées dans le modèle.

Ces règles seront décrites plus en détails dans la définition sémantique de TVL-P et dans le chapitre suivant traitant de l'élaboration d'un outil de vérification de ces règles.

7.8 Sémantique

La définition sémantique de TVL-P repose, tout comme TVL (cfr 4.1), sur une définition abstraite et respecte les recommandations de Harel et Rumpe [18].

Cette section vise donc à définir une syntaxe formelle (domaine syntaxique), un domaine sémantique et une fonction sémantique permettant de définir la sémantique d'un produit exprimé dans le langage abstrait $\mathcal{L}_{TVLP}$.

7.8.1 Domaine syntaxique

La syntaxe abstraite de TVL-P, notée $\mathcal{L}_{TVLP}$, est inspirée de la syntaxe abstraite de TVL. Les éléments de la version normalisée de TVL-P sont associés un à un aux éléments de cette syntaxe abstraite. Le processus de normalisation d'un produit TVL-P est décrit en 7.9.

Le domaine syntaxique de TVL-P détermine tout ce qui peut être écrit en utilisant les termes et règles syntaxiques de $\mathcal{L}_{TVLP}$.

$\mathcal{L}_{TVLP}$ est défini comme ceci : $d \in \mathcal{L}_{TVLP}$ un est n-tuple (C', r', A, ρ, μ) tel que :

- C' : ensemble de clones sélectionnés, tels que $C' \subseteq C$, où :
  - F : ensemble (non vide) de features
  - C : ensemble de clones tel que : *« Si un clone est défini comme un tuple(feature, children) où children est un multiset de clones, alors le set de tous les clones possibles est C, tel que* $C \subseteq F \times \text{powerbag}(C)$ *»* (Michel et al. [22])
- $r' \in C'$ est le clone racine
- A est un set de tous les attributs
- $\rho : A \rightarrow F$ est une fonction qui retourne le feature auquel est attaché un attribut
- $\mu : A \times C' \rightarrow N \cup Q \cup \{true, false\} \cup String$ est une fonction qui retourne la valeur d'un attribut dans une instance (clone) d'un feature.

Remarque : La structure hiérarchique est définie par les clones, puisqu'un clone est défini sous la forme Clone ::= (Feature, Children) où children sous les clones « enfants » de ce clone).

7.8.2 Domaine sémantique

Dans le cas de TVL, le domaine sémantique inclut tous les produits possibles dérivés de tous les feature diagrams possibles exprimés à partir du domaine syntaxique de la syntaxe abstraite de TVL. Suite à l'introduction du clonage, ce domaine sémantique est le set de tous les clones possibles, puisqu'un clone est défini comme un feature et son mutltiset d'enfants (cfr Michel et al. [22]).

Le domaine sémantique de TVL-P, noté S_{TVLP}, se définit de façon similaire à celui de TVL. La différence entre un modèle TVL et un produit TVL-P est que le modèle exprime les produits possibles (parfois un seul ou même aucun) alors que TVL-P exprime dans le plupart des cas un produit bien particulier issu d'un modèle, mais nous avons vu que la définition peut aussi être partielle.

Les domaines de TVL et TVL-P sont donc comparables, ils expriment tout deux des sets de clones.

Une définition formelle, basée sur les définitions des clones et du domaine sémantique de TVL (cfr Michel et al. [22]) ,est donc celle-ci:

« Si C est un set de tous les clones possibles comme défini en 7.8.1, le domaine sémantique est défini comme $\mathcal{S}_{TVLP} = C$ *»* (adaptation de la définition du domaine sémantique de TVL de Michel et al.)

7.8.3 Fonction sémantique

Selon les recommandations de Harel et Rumpe [18], par analogie avec la fonction sémantique de TVL (Michel et al. [22]), la fonction sémantique associe des éléments du domaine syntaxique au domaine sémantique. Cette fonction est notée comme ceci :
$\mathcal{M}_{TVLP} : \mathcal{L}_{TVLP} \rightarrow \mathcal{S}_{TVLP,}$ où $\mathcal{L}_{TVLP}$ et $\mathcal{S}_{TVLP}$ représentent respectivement les domaines syntaxique (cfr 7.8.3) et sémantique (cfr 7.8.4) de TVLP.

Cette fonction permet donc de donner la signification d'un produit exprimé en $\mathcal{L}_{TVLP}$. Ce produit va correspondre, s'il est valide, à un clone appartenant à l'ensemble de clones C (cfr 7.8.1).

Un produit exprimé par $\mathcal{L}_{TVLP}$ est valide par rapport à un modèle de $\mathcal{L}_{TVL}$ s'il correspond à un produit valide dérivé de ce modèle de $\mathcal{L}_{TVL}$.

Cette fonction est définie comme ceci : Soit d $\in \mathcal{L}_{TVLP}$ et m $\in \mathcal{L}_{TVL}$ tel que $\mathcal{D}$(d, m) :
$\mathcal{M}_{TVLP}$(d) est un produit p si ∃ p tel que p $\in \mathcal{M}_{TVL}$(m), où p ∈ C, sinon $\mathcal{M}_{TVLP}$(d) = {}

$\mathcal{D}$(d, m) est vrai si le produit d $\in \mathcal{L}_{TVLP}$ est dérivé du modèle m $\in \mathcal{L}_{TVL}$ où :

- d est défini en 7.8.1 comme le tuple (C', r' , A, ρ, μ)
- m est défini dans « Syntax & Semantic of TVL » [20] et adapté par Michel et al [22] comme le tuple (F, r, DE, ω, λ, A, ρ, τ , Φ) (cfr 5.2.6 pour rappel)

Cela signifie que $\mathcal{D}$(d, m) est vrai si :

- C' ⊆ C ⊆ F × powerbag(C)
- r' est le tuple (r, children) où r ∈ F est la racine de m et children est le multiset des clones enfants de r'
- A et ρ du tuple définissant d sont identiques à ceux présents dans le tuple de m.
- Les valeurs fournies par μ respectent τ pour chaque attribut

Remarques :

- Un clone est un tuple composé d'un feature dont il est une instance et un multiset de clones enfants. Vu cette définition récursive, un seul clone peut donc représenter une hiérarchie de clones et donc un produit.
- Vu que p doit être tel que p $\in \mathcal{M}_{TVL}$(m), ceci garanti que $\mathcal{M}_{TVL}$ doit être respecté, ce qui implique que la structure hiérarchique définie par les clones doit respecter DE et que les contraintes Φ doivent être vérifiées.

- Dans cette définition, les attributs de A sont des attributs simples. La sémantique des attributs composés de TVL-P est donc obtenue en normalisant ces attributs (les convertir en un set d'attributs simples), ils correspondent alors aux attributs du langage abstrait et leur sémantique. Cette définition impliquera donc un choix lors de la validation des attributs d'un produit et du modèle, doivent-ils être normalisés ou pas ? Ce cas sera traité dans le chapitre suivant à propos de la validation d'un produit par rapport à un modèle (cfr 8.3.2).

7.9 Normalisation

Vu la syntaxe relativement simple de TVL-P par rapport à celle de TVL, le processus de normalisation de TVL-P est réduit.

Parmi la liste d'opérations du processus de normalisation d'un modèle TVL (cfr « Syntax & Semantic of TVL » [20]), deux ont une étape équivalente lors de la normalisation d'un produit TVL-P :

- Elimination des directives « include » permettant d'inclure le contenu d'un autre fichier, en les remplaçant par le contenu des fichiers référencés.

- Regrouper les définitions de clones dispersées dans plusieurs blocs de définitions (cas des définitions en extension).

- Normaliser les attributs composés, ce qui signifie les convertir en attributs simples de façon similaire à la normalisation des attributs de TVL (cfr [20]).
  Par exemple coord {x is 1 ; y is 0} devient coord_x is 1 ; cord_y is 0; dans la version normalisée.

Les autres étapes de la normalisation de TVL n'ont pas d'équivalence dans TVL-P puisque les éléments auxquels elles se rapportent n'ont pas d'équivalent dans TVL-P.

7.10 Limites et travaux futurs.

Ce chapitre a défini la syntaxe et la sémantique de TVL-P : un langage de configuration inspiré de TVL dans le but d'exprimer des configurations de modèle TVL.

La définition d'un nouveau langage dérivé de TVL a été préférée à l'utilisation du langage de configuration PSL issu du projet HATS afin d'offrir une plus grande cohérence par rapport à TVL. Les possibilités offertes par PSL (sélectionner les features et assigner des valeurs aux attributs) sont d'ailleurs aussi permises par TVL-P. La description d'un bloc d'initialisation permise en PSL n'a de sens que dans le contexte du projet HATS, pas dans le celui de TVL.

TVL-P étant dérivé de TVL, les constructions autorisées dans TVL-P dépendent de celles autorisées dans TVL. Une modification des possibilités offertes dans TVL impliquerait donc une révision de TVL-P.

TVL-P permet d'instancier des features dans le produit et d'assigner des valeurs à leurs attributs. Il a été décider de ne pas permettre d'introduire de définitions de contraintes ou de relations entre les instances de features et leurs attributs dans la définition des produits. Cette limitation pourrait être revue à l'avenir en cas de besoins rencontrés dans des cas industriels.

Afin d'être valide, l'expression d'une configuration doit respecter une grammaire et les quelques règles du langage non garanties par la grammaire. Ceci peut être vérifié facilement par un outil tel qu'un simple parser de langage basé sur un analyseur syntaxique auquel un module d'analyse sémantique est ajouté afin de vérifier des règles sémantiques non vérifiées par la grammaire.

Mais cette condition n'est pas suffisante. Un produit exprimé en TVL-P doit également être valide par rapport à un modèle, exprimé en TVL.

Le chapitre suivant traitera de la validation des produits exprimés dans le langage TVL-P, à l'aide d'outils.

8. Validation d'un produit TVL-P

Ce chapitre va décrire un système destiné à analyser des produits exprimés en TVL-P et les valider par rapport à un modèle exprimé en TVL.

Ce chapitre n'est pas une documentation technique de ce système, il ne contiendra donc pas une description détaillée de l'implémentation, mais les motivations de l'élaboration de ce système, une présentation du domaine, des exigences et des spécifications. Ainsi qu'un présentation de l'architecture et des principaux choix d'implémentation.

Ce système a été implémenté dans le cadre de ce mémoire.

8.1 Motivations

Le chapitre 4 a présenté le langage TVL destiné à exprimer des modèles, ainsi qu'un parser afin d'analyser ces modèles. Le chapitre 5 a ensuite adapté TVL et le parser suite à l'introduction du clonage.

Le chapitre précédent a défini le langage TVL-P, destiné à exprimer des configurations de produits issus de modèles TVL.

La suite logique de ces travaux est donc la conception d'un outil de validation des configurations exprimées en TVL-P.

En effet, une des principales motivations de la conception du langage TVL-P est de pouvoir exprimer des configurations de produits de manière simple et intuitive mais aussi d'offrir des outils de validations similaires à ceux utilisés pour valider les modèles TVL.

Cet outil devra donc vérifier qu'une configuration exprimée en TVL-P est syntaxiquement correcte, mais aussi qu'elle correspond bien au modèle TVL dont elle est dérivée.

8.2 Domaine et Exigences

Dans le cadre d'un système où l'utilisateur souhaite exprimer des configurations de modèles en utilisant le langage TVL-P et valider ces configurations, voici les contraintes du domaine et les exigences.

Contrainte du domaine

Un utilisateur souhaitant valider un produit TVL-P par rapport à un modèle doit disposer de la description valide de ce modèle dans le langage TVL.

Exigences

- Vérifier la syntaxe du produit TVL-P (cfr grammaire 7.6)
- Vérifier que le produit TVL-P respecte les contraintes sémantiques de TVL-P indépendantes d'un modèle (cfr 7.7)
- Vérifier la validité du produit TVL-P par rapport au modèle TVL (cfr définition sémantique 7.8.3).

- Les outils utilisés par le système devront être libres (ou appartenir aux FUNDP) et portables.
- Les processus et outils utilisés lors de la validation d'un produit TVL-P devront être proches des outils et processus utilisés lors de la validation d'un modèle TVL. Ils seront, comme pour TVL, invocables via leur API afin de faciliter leur intégration avec les outils existants et les outils futurs.

8.3 Fonctionnalités

8.3.1 Liste des fonctionnalités

Afin de répondre aux exigences exprimées ci-dessus, un parser TVL-P est nécessaire, et doit fournir les fonctionnalités suivantes :

- Analyse syntaxique du produit TVL-P défini par la grammaire de TVL-P (cfr 7.6).
- Vérification des règles non vérifiées par la grammaire (cfr 7.7)
- Validation du produit par rapport au modèle selon les règles de validité d'un produit (cfr sémantique de TVL de Michel et al. et définition sémantique de TVL-P du point 7.8.3). **Attention au cas des attributs (cfr 8.3.2) et des contraintes (cfr 8.3.3)**
- Rapport à propos du résultat de la validation et du contenu du code TVL-P analysé.

8.3.2 Cas particulier : Validation des attributs

Dans le langage abstrait, la définition des attributs impose que ceux-ci soient des attributs simples (cfr 7.8.3). Au niveau abstrait, la comparaison d'un produit et d'un modèle se fait donc en utilisant des attributs simples. Mais au niveau concret de TVL et de TVL-P, les attributs peuvent être composés.

La comparaison des attributs d'un produit et d'un modèle devrait-elle avoir lieu sans normalisation ni du produit ni du modèle ou avec normalisation du produit et du modèle afin de comparer uniquement des attributs simples ?

Si les attributs sont simples dans le produit et dans le modèle ou s'ils sont composés selon la même composition dans le produit et dans le modèle, normaliser ou pas n'a aucune importance.

Par contre, si l'un est composé et l'autre pas, la première solution est plus stricte puisque dans ce cas, l'attribut composé est différent des attributs simples, alors que l'attribut composé serait structurellement identique au set des attributs simples correspondants en cas de normalisation.

Du point de vue de l'utilisateur, TVL-P devrait-il autoriser l'encodage d'un attribut composé dans un produit alors que le modèle est défini avec des attributs simples ou inversement ? Certains utilisateurs pourraient souhaiter cette souplesse alors que d'autres préféreraient plus de rigueur en prétextant que si la modèle a été défini en utilisant des attributs composés, c'est pour marquer les liens logiques entre différents attributs et que dès lors, autoriser leur utilisation sous forme simple cacherait ces liens entre les sous-attributs.

Un autre problème pratique se pose. Si l'utilisateur utilise des attributs simples dans le produit alors que le modèle utilise un attribut composé, la validation avec normalisation

ne pourrait fonctionner que si l'utilisateur connaît la manière dont les attributs composés sont transformés en attributs simples lors de la normalisation.

En conclusion, la solution « avec normalisation » est plus souple, mais à condition que l'utilisateur connaisse en partie le fonctionnement de l'algorithme de normalisation. Cette souplesse serait donc rarement possible dans les cas pratiques. Cette première version de la vérification de validité du produit n'effectuera pas de normalisation avant la validation es attributs. Bien entendu, ce choix sera à réévaluer à la lumière des futurs cas pratiques rencontrés.

8.3.3 Validation des contraintes

Selon la définition sémantique (cfr 5.2.6) de la validité d'un produit, un produit est valide si les contraintes du modèle (cfr 4.2.4) sont vérifiées par le produit.

Mais vu les modifications de la syntaxe et de la sémantique des expressions des contraintes « cross-tree » présentées au chapitre 6, la première version de ce parser ne validera pas ces contraintes.

En effet, ces propositions de modifications n'ont pas encore été implémentées dans le langage TVL.

De plus, la validation de telles contraintes nécessite l'adaptation d'un solver SMT, travail en cours par Raphael Michel.

8.3.4 Principes d'utilisation du parser

De façon similaire au parser TVL, ce parser doit être utilisable via son API. Il reçoit en entrée la description d'une configuration exprimée en TVL-P ainsi que le modèle de référence exprimé en TVL. Il exécute alors les analyses lexicale, syntaxique et sémantique et indique le résultat de ces analyses :

- Produit mal formé : erreur lors de l'analyse lexicale ou syntaxique
- Produit bien formé mais non valide sémantiquement (soit une règle indépendante du modèle n'est pas vérifiée, soit le produit ne respecte pas le modèle)
- Produit valide.

Une option permet lors de l'appel de préciser si l'utilisateur souhaite un rapport décrivant le contenu de l'AST et de la table des symboles.

En cas d'erreur au cours de l'analyse, l'erreur rencontrée sera transmise à l'utilisateur.

8.4 Architecture et choix techniques.

L'architecture du parser de TVL-P est une architecture classique d'un parser de langage comme défini au chapitre 3. Elle sera inspirée de l'architecture du parser de TVL décrite au chapitre 4.

Par cohérence avec le parser TVL, le parser TVL-P est écrit en Java et utilise les outils JFlex [29] et CUP [30].

Les principaux éléments de ce parser sont :

- Un analyseur lexical, généré par l'outil JFlex sur base de la spécification des unités lexicales rencontrées dans TVL-P.
- Un analyseur syntaxique généré par l'outil CUP sur base de la grammaire au format CUP de TVL-P.
- Un analyseur sémantique permettant de parcourir l'AST généré par l'analyseur syntaxique afin de générer la table des symboles de TVL-P et de vérifier la validité du produit par rapport au modèle.
- Le parser TVL appelé par le parser TVL afin d'analyser le modèle et de générer sa représentation interne, utilisée par l'analyseur sémantique de TVL-P lors de la vérification de la validité du produit par rapport au modèle.

La figure ci-dessous (8.1) illustre ces différents éléments.

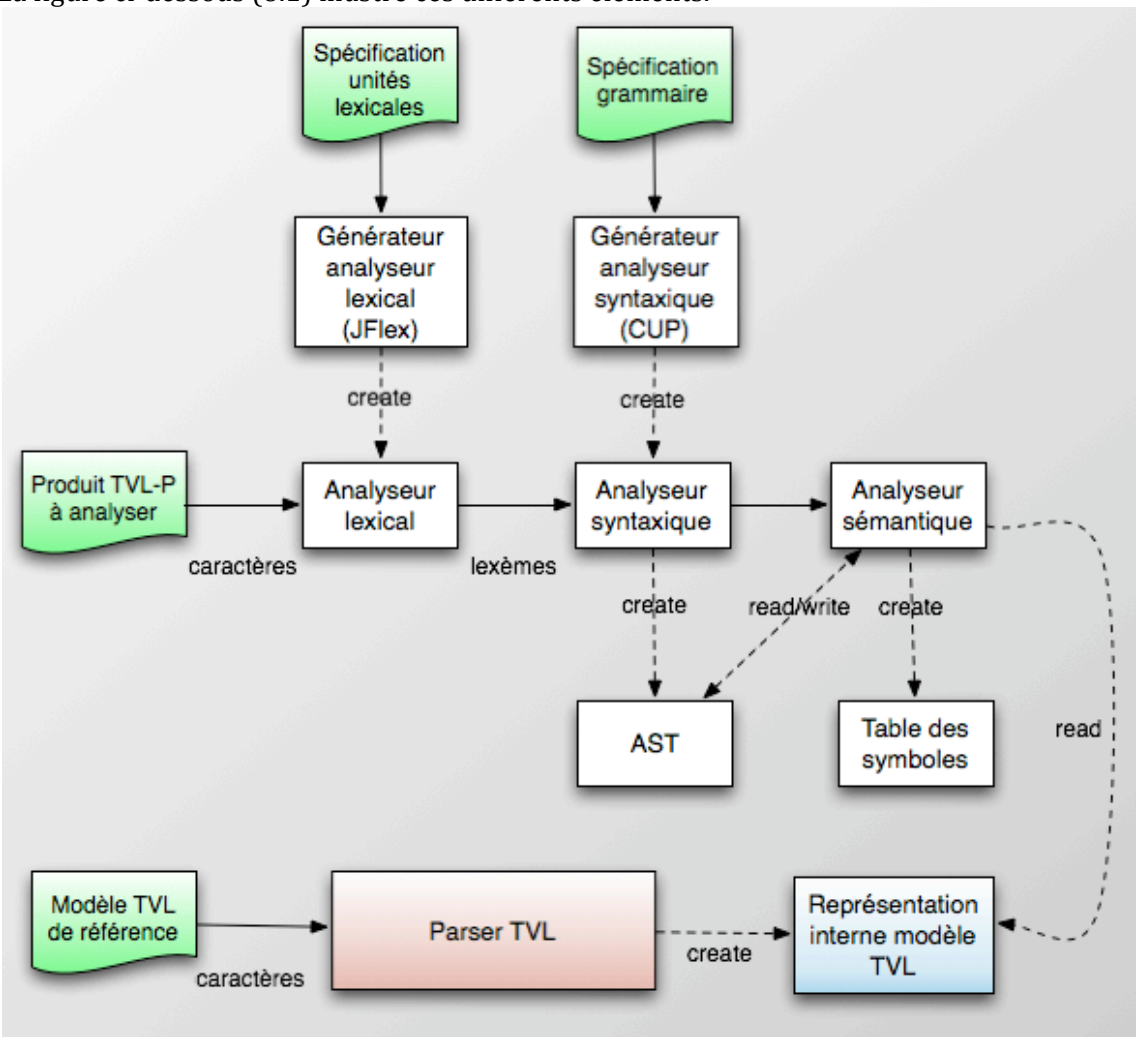


Figure 8.1 Structure du parser TVL-P

8.5 Conception du parser

Cette section décrit l'enchainement des différentes étapes de l'analyse d'un fichier TVL-P et décrit les composants utilisés. Il ne s'agit pas d'une description complète et détaillée de l'implémentation de ces composants, mais d'une présentation des principaux choix tels que la structure de l'arbre syntaxique abstrait et de la table des symboles.

8.5.1 Flux général d'une analyse

Afin de permettre l'appel du parser, une classe Launcher est ajoutée et représente le point d'entrée du parser.

Cette classe dispose d'une méthode recevant en paramètres le produit TVL-P et le modèle TVL. Elle peut alors déclencher les analyses lexicale et syntaxique du produit TVL-P.

Ensuite, si le produit est correct indépendamment du modèle, elle peut transmettre le modèle au parser TVL et en demander l'analyse afin d'obtenir la représentation interne de ce modèle.

L'analyseur sémantique de TVL-P peut alors être invoqué afin de valider le produit par rapport au modèle, en se basant sur la représentation interne du produit TVL-P (son AST) et la représentation du modèle TVL (table des symboles issue du parser TVL).

Si le Launcher reçoit un paramètre demandant un rapport à propos du contenu du produit, il invoquera les méthodes de rapport définies sur l'AST et/ou la table des symboles puis exportera les XMLs générés dans des fichiers ou dans la console (selon que les paramètres décrivant les paths des fichiers dans lesquels exporter le contenu de l'AST et/ou de la table des symboles soient spécifiés ou non).

8.5.2 Analyse syntaxique et Arbre syntaxique abstrait

La grammaire du langage TVL-P (cfr 7.6) est complétée afin d'y ajouter les actions permettant d'indiquer comment l'AST doit être construit. Le parser ainsi généré par CUP pourra invoquer ces actions afin de construire l'AST au fur et à mesure qu'il analyse le code TVL-P. Avant de décrire le principe de ces « actions », voici la description de l'arbre syntaxique :

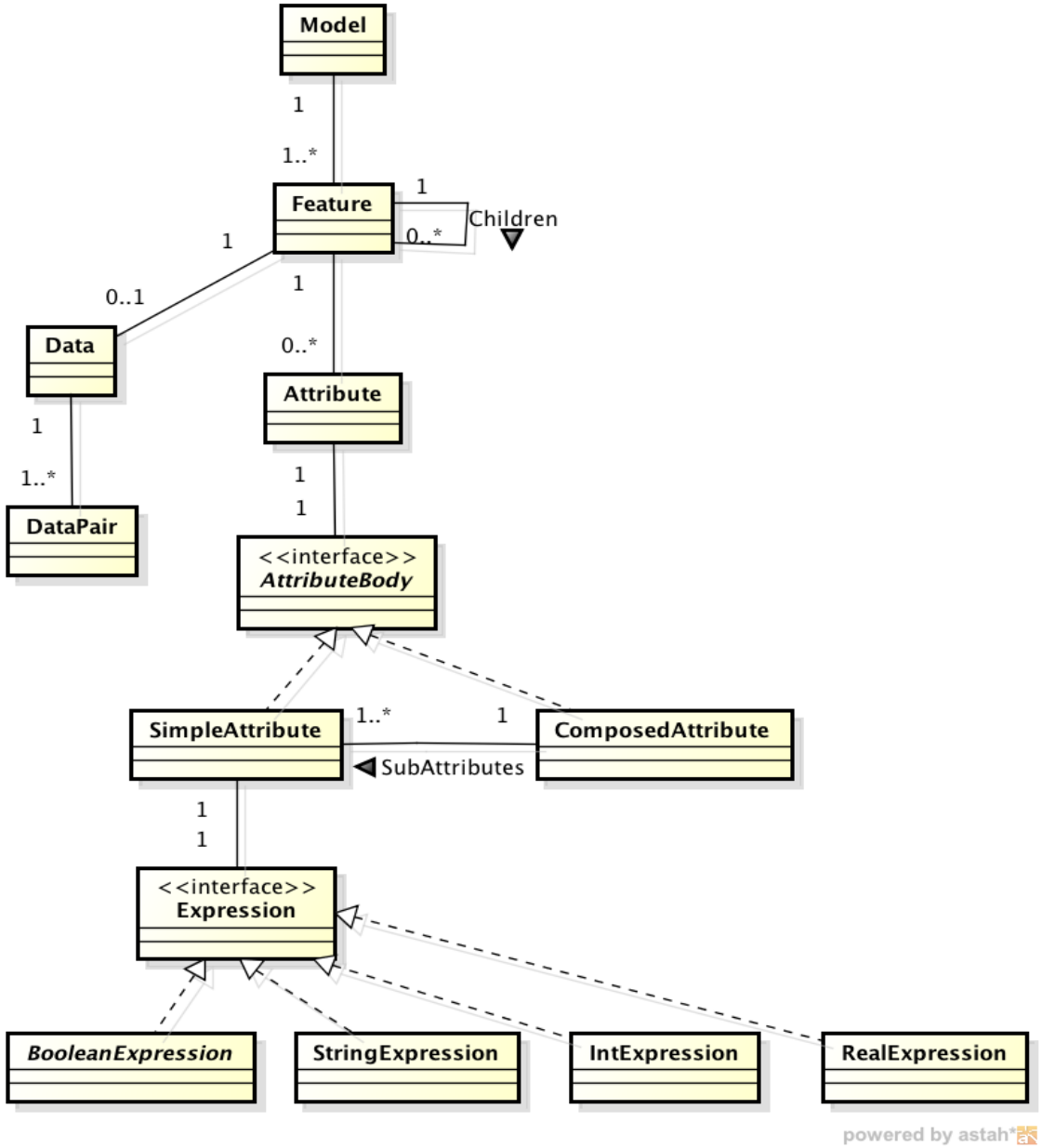


Figure 8.2 Structure de l'AST

La grammaire de TVL-P est une grammaire LALR, l'analyseur généré est donc un analyseur de type ascendant. La structure de l'arbre syntaxique abstrait est proche de celle de la grammaire.

Dans le schéma de l'AST, la description d'une configuration exprimée en TVL-P (« Model ») contient un nombre de déclarations de clones de feature (« Feature ») pouvant contenir

des attributs (« Attribut »), soit simples (« SimpleAttribute ») soit composés (« ComposedAttribute »). Un attribut composé est une collection d'attributs simples et un attribut simple est associé à une expression (« Expression ») pouvant être de différents types.

Nous pouvons à présent nous intéresser au principe général d'utilisation des actions. Une action est une suite d'instructions exécutées lorsque l'analyseur reconnaît une règle de la grammaire. Puisque la structure de la grammaire est comparable à la structure de l'AST, le parser va créer les objets de l'AST au fur et à mesure qu'il reconnaît les règles de la grammaire.

Par exemple, il reconnaît une expression, il génère donc un objet de type « Expression » contenant la valeur lue. Il reconnaît ensuite un attribut composé d'un nom et de l'expression, il peut alors générer un objet de type « Attribute » contenant l'objet de type « Expression » généré lors de la rencontre de l'expression.

La traduction de la grammaire en un fichier CUP nécessite quelques adaptations :

- les objets retournés par les actions décrites dans le fichier CUP doivent être typés, ce qui implique la définition de super-types en cas de règles composées d'alternatives retournant des objets de types différents
- une règle dans le fichier CUP ne peut pas utiliser ()* pour représenter une collection, une règle intermédiaire récursive est donc nécessaire afin de représenter la collection, sous la forme LIST ::= ELEMENT LIST

Des interfaces devront être ajoutées afin de fournir des super-types et des types seront aussi ajoutés afin de représenter les collections de valeurs construites par des règles intermédiaires. Ces nouveaux types et interfaces ne sont pas présents dans l'arbre mais sont utilisés lors de la construction de celui. Vu qu'il s'agit de détails d'implémentation, nous ne nous attarderons pas sur le sujet.

8.5.3 Analyse sémantique et table des symboles

Rôles de l'analyse sémantique

L'analyse sémantique consiste à vérifier les contraintes de TVL-P non vérifiées par l'analyse syntaxique et vérifier la validité du produit par rapport au modèle, ce qui concrètement consiste à vérifier les conditions suivantes (issues de la définition sémantique de TVL et TVL-P) :

- Le nom de feature existe-t-il dans le modèle ?
- Le clone a-t-il une seule définition de sa décomposition ?
- La hiérachie de features définie dans le modèle est-elle respectée ?
- Les cardinalités de groupe sont-elles respectées ?
- Les cardinalités de feature sont-elles respectées ?
- L'attribut est-il défini dans le modèle
- La valeur assignée à un attribut est-elle compatible avec le type de l'attribut et ses restrictions de valeurs du domaine ?
- L'attribut peut-il se voir attribué une valeur et en a-t-il reçu une seule ?

L'analyseur sémantique doit également générer une table des symboles contenant les informations à propos des clones présents dans le produit : leur feature associé, leur collection de clones enfants et les valeurs assignées à leurs attributs. Cette table de symbole devra exposer ces informations.

L'analyse sémantique et la création de la table des symboles peuvent s'opérer simultanément, ce qui permet d'arrêter le processus directement en cas d'erreur sémantique.

Rôles de la table des symboles

La table des symboles doit stocker les informations relatives aux clones présents dans la configuration, elle doit exposer des méthodes permettant de :

- Obtenir les informations (feature, parents, enfants, attributs) à propos d'un clone donné à partir du nom qualifié du clone ou de son alias.
- Retourner le clone racine
- Modifier les informations à propos des clones.
- Ajouter un clone dans la table des symboles
- Obtenir un rapport XML décrivant les éléments qu'elle stocke.

Pour chaque clone, les éléments suivants doivent être stockés :

- Identifiant du feature dont le clone est une instance
- Alias donné au clone
- Parent(s) du clone (un clone partagé à plusieurs parents)
- Liste des clones enfants
- Liste des attributs

Structure de la table des symboles

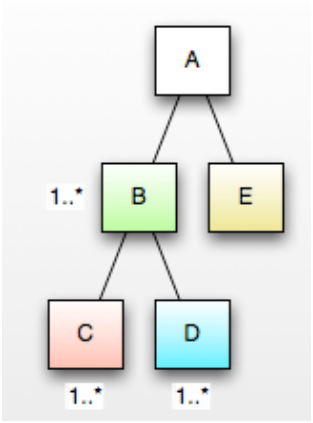
La structure interne de cette classe relève des choix d'implémentation, mais ces choix peuvent faciliter la vérification des conditions de validité du produit.

Afin de vérifier les cardinalités de groupe et de feature, le stockage des clones enfants doit permettre d'identifier et de compter les features représentés et les clones présents pour chaque feature. La solution proposée consiste donc à utiliser une Map de listes afin de stocker les clones enfants.

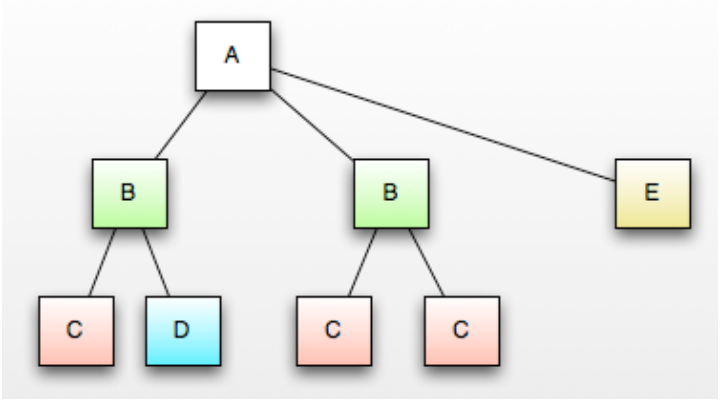
Chaque clone (« nommé clone courant ») contiendra une Map de listes contenant une entrée par feature rencontré. Chaque entrée référencera alors une liste contenant les enfants du clone courant dont le feature associé est celui associé à l'entrée courante.

Voici un exemple afin d'illustrer cette Map de listes, le modèle et le produit sont exprimés sous forme de feature diagram pour en faciliter la compréhension :

Le modèle est constitué d'une racine « A » décomposée en sous-features « B » et « E ».
« B » est clonable et décomposé en C et D, eux aussi clonables :



Voici un produit correspondant au modèle ci-dessus, « A » dispose de deux clones de « B », composés respectivement de clones de {C, D} et {C, C}

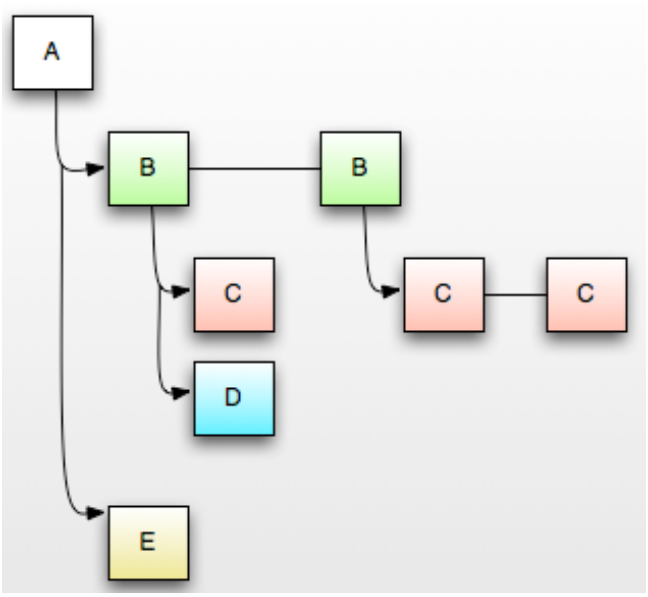


La racine de ce produit est un clone de « A », ce clone sera donc l'élément racine de la table de symbole.

Les clones enfants du clone de « A » sont de deux types : « B » et « C ». La Map de liste du clone de « A » aura donc deux entrées :

- l'une pour les clones du feature « B », elle référencera une liste contenant les deux clones de « B » :
  - le premier contiendra une Map de listes avec des entrées pour « C » et « D », référençant respectivement une liste contenant un clone de « C » et une liste contenant un clone de « D ».
  - le second contiendra une Map de listes avec une entrée pour « C » qui référence une liste contenant deux clones de « C ».
- l'autre pour les clones du feature « E ».

Le schéma de la table des symboles correspondante est celui-ci :



Cette structure permet de classer les clones par type de feature.

Pour obtenir la liste des features présents parmi les enfants d'un clone, il suffit de consulter la liste des entrées de sa Map de listes d'enfants.

Le nombre de ces entrées est donc le nombre à prendre en compte lors de la vérification des cardinalités de groupe puisque celles-ci se vérifient localement et en comptant le nombre de features représentés.

La vérification des cardinalités d'un feature « f » nécessite de connaître le nombre de clones de ce feature « f » sous chaque clone du feature « p », parent de « f ». Ce nombre est obtenu en comptant le nombre de clones dans une liste de clones, puisque chaque liste contient tous les clones d'un feature donné sous un parent donné.

8.6 Limites et développements futurs.

La validation d'un produit TVL-P par rapport à un modèle TVL est donc possible en utilisant un parser semblable au parser TVL du point de vue de son architecture et des outils utilisés.

Le parser TVL est utilisé par le parser TVL-P afin d'analyser le modèle TVL de référence.

Toutefois, la nouvelle version de la syntaxe et de la sémantique des contraintes « cross-tree », adaptée au clonage, n'est pas encore gérée dans ce parser TVL-P pour deux raisons :

- Cette nouvelle version de syntaxe et sémantique n'est pas encore implémentée dans le parser TVL.
- Un solver gérant le clonage est nécessaire et doit être intégré à ce parser TVL.

Lorsque ces deux éléments seront réalisés, le parser TVL-P pourra alors être adapté afin de vérifier si la configuration décrite en TVL-P respecte les contraintes « cross-tree » du modèle de référence.

Des développements futurs pourraient améliorer l'utilisation de TVL-P au niveau ergonomique.

En effet, l'utilisation du parser en ligne de commandes est peu ergonomique, mais la possibilité de l'utiliser via son API permet de l'intégrer facilement à d'autres outils.

Nous pourrions alors imaginer un éditeur TVL-P offrant des fonctions d'aide à l'encodage (« auto-completion », « syntax highlighting ») et permettant de valider le code TVL-P en utilisant le parser TVL-P.

Conclusion

La première partie de ce mémoire a rappelée la notion de Software Product Line, les processus qui la composent, ses avantages tels que la réduction des coûts et des délais de production mais aussi les difficultés de sa mise en œuvre.

La variabilité des SPLs a été introduite, ainsi que les feature diagrams permettant de représenter cette variabilité.

Une version textuelle de feature diagrams a été présentée : TVL. Ses avantages par rapport aux versions graphiques ont été présentés, ainsi que l'outil de validation de modèles TVL, basé sur les techniques classiques d'analyse de langages, qui ont été brièvement rappelées.

Nous avons vu une limitation de la version de TVL antérieure à ce mémoire : celle-ci n'intègre pas le clonage, bien que la sémantique du clonage ait déjà été définie et que plusieurs versions de feature diagrams permettent le clonage. Pour rappel, le clonage signifie la capacité de pouvoir instancier dans un produit plusieurs occurrences d'un feature du modèle.

La première contribution de ce mémoire, décrite au chapitre 5, a donc consisté à intégrer ce principe de clonage à TVL, sur base de la syntaxe abstraite et de la sémantique du clonage.

La syntaxe concrète de TVL, définie par sa grammaire, a donc été adaptée afin de permettre la spécification de cardinalités de feature indiquant combien de fois un feature peut être cloné sous une instance de son parent. Nous avons vu que ces cardinalités de feature peuvent uniquement être déclarées lors de la première déclaration d'un feature et qu'elles ne peuvent pas être déclarées ni pour un feature racine ou optionnel (« opt »), ni pour un feature partagé ou un de ses parents.

La structure de la représentation interne (AST et table de symboles) a été adaptée afin de mémoriser et valider les cardinalités de feature.

L'implémentation de l'outil de validation des modèles TVL a été adaptée selon les modifications citées ci-dessus.

Toutefois, un point important restait à définir : l'expression de contraintes « cross-tree » de TVL. En effet, l'ancienne version de ces contraintes n'était plus valide suite à l'introduction du clonage, car elles manipulaient des features instanciés une seule fois, et non de multiples instances de ces features.

Le chapitre 6 a donc proposé une seconde contribution : une nouvelle syntaxe et une nouvelle sémantique de ces expressions basées sur des quantificateurs mathématiques et sur une navigation dans le graphe du produit permettant de sélectionner des ensembles d'instances (clones) et des collections de valeurs. La syntaxe abstraite, la sémantique ainsi que la syntaxe concrète de ces expressions ont été définies, mais pas encore implémentées dans un outil de validation de modèle TVL. Ces propositions de nouvelles syntaxe et sémantique devront être confrontées à d'autres cas industriels plus complexes et volumineux afin d'évaluer et d'adapter si nécessaire les solutions proposées.

Enfin, la troisième contribution a défini une extension du langage TVL, afin d'exprimer des configurations de produits. Cette extension, nommée TVL-P, a été définie par le chapitre 7, au niveau syntaxique et sémantique. La syntaxe de TVL-P est dérivée de la syntaxe de TVL, elle offre donc les mêmes avantages que TVL, mais au niveau de l'expression des configurations de produits.

De façon analogue à TVL, un outil est nécessaire afin de vérifier la validité des configurations exprimées en TVL-P. Ces configurations doivent donc être vérifiées au niveau syntaxique et sémantique, mais aussi par rapport au modèle TVL dont la configuration est dérivée. Le chapitre 8 a donc présenté les exigences d'un tel outil ainsi que sa conception.

Cet outil a été implémenté, mais dans un premier temps, sans la gestion des contraintes « cross-tree », puisque celle-ci ne sont pas non plus implémentées dans le parser TVL et nécessiteraient l'utilisation d'un solver capable de gérer le clonage, ce qui n'était pas le cas du solver utilisé avant l'élaboration de ce mémoire.

La prochaine étape de l'implémentation des outils de validations de TVL et de TVL-P pourrait donc être l'implémentation de la vérification syntaxique et sémantique des contraintes « cross-tree » dans le parser TVL et l'intégration d'un solver capable de gérer le clonage aux parsers TVL et TVL-P afin de vérifier la satisfaisabilité des modèles TVL et la validité des configurations exprimées en TVL-P par rapport à leur modèle.

Plusieurs types d'outils pourraient être développés afin de faciliter l'utlisation de TVL et TVL-P.

Par exemple, des éditeurs de fichiers TVL / TVL-P offrant des fonctionnalités d'aide à l'encodage (« auto-completion », « syntax highlighting ») et de validation des modèles et des configurations par des appels aux parsers TVL / TVL-P via une interface graphique et non en ligne de commande.

Bien que le but de TVL et de TVL-P est de permettre aux utilisateurs de décrire des modèles et des configurations de manière textuelle, des outils de configurations pourraient être développés afin de fournir une interface graphique de configuration aux utilisateurs. Celle-ci pourrait alors traduire les choix exprimés par l'utilisateur par l'intermédiaire de l'interface graphique en code TVL-P. Un projet de mémoire sous la responsabilité du professeur Heymans va d'ailleurs dans ce sens en étudiant la possibilité de générer des interfaces graphiques de configuration à partir de modèle TVL. Le langage TVL-P et son parser pourraient permettre de valider et de sauvegarder la configuration générée par les choix de l'utilisateur.

Bibliographie

[1] K. Pohl, G Böckle, F. van der Linden. « Software Product Line Engineering : Foundations, Principles and Techniques », 2005, Springer-Verlag.

[2] P.C. Clements, L. Northrop, « Software Product Lines : Practices and Patterns. » Addison-Wesley, 2001.

[3] Kyo Chul Kang. FODA : « Twenty Years of Perspective on Feature Modelling. » In Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 2010.

[4] A. Classen, P. Heymans and P.Y. Schobbens. « What's in a feature : A Requirements Engineering Perspective. » PReCISE Research Center, Faculty of Computer Science, University of Namur, 2008.

[5] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. « Feature-oriented domain analysis (FODA) feasibility study. » Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[6] Bosch, J.: « Design and use of software architectures: adopting and evolving a product-line approach ». ACM Press/Addison-Wesley, New York, USA (2000)

[7] Chen, K., Zhang, W., Zhao, H., Mei, H.: « An approach to constructing feature models based on requirements clustering ». In: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05). (2005) 31-40

[8] Benavides, P. T. Martın-Arroyo, and A. R. Cortés, "Automated reasoning on feature models," in Proceedings of CAiSE'05, 2005.

[9] K. Czarnecki, S. Helsen, and U. Eisenecker. « Staged configuration using feature models ». In Software Product Lines: Third International Conference, SPLC 2004, pages 266–283, 2004. Springer-Verlag.

[10] K Czarnecki and C. H. P. Kim. "Cardinality-based feature modeling and constraints : a progress report". In International Workshop Factories at OOPSLA'05,San Diego, California, USA, 2005. ACM.

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. « Formalizing cardinality-based feature models and their specialization ». In Software Process : Inprovement and Practice, Special issue on Variability : Software and management, Vol 10, Issue 1, pages 7-29, 2005. doi: 10.1002/spip.213

[12] Riebisch, M., Böllert, K., Streitferdt, D. and Philippow, I. (2002). "Extending feature diagrams with UML multiplicities"

[13] Riebisch M. « Towards a More Precise Definition of Feature Models ». In Modelling Variability for Object-Oriented Product Lines. Pages 64-76, Riebisch, M.; Coplien, J. O. & Streitferdt, D. *(ed.)*, 2003

[14] Kwanwoo. Lee, Kyo. C. Kang, Jaejoon Lee. “Concepts and Guidelines of feature Modeling for Product Line Software Engineering”. In Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), pages 62-77, Springer-Verlag, 2002.

[15] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, Koen G. Langendoen, « Compilateurs, » Dunod, 2002. ISBN 2-10-005887-8.

[16] Université de Bretagne Occidentale. Compilation, « Théorie des langages ». Université de Bretagne Occidentale, 2004.

[17] P.Y. Schobbens. « Syntaxe et sémantique des langages de programmation », Faculté d’informatique, FUNDP, Namur.

[18] D. Harel and B. Rumpe, “Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff,” Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, Tech. Rep., 2000.

[19] A.Classen, Q. Boucher, and P. Heymans. « A text-based approach to feature modelling: Syntax and semantics of TVL » (in press). Science of Computer Programming, Special Issue on Software Evolution, 2010. doi:10.1016/j.scico.2010.10.005.

[20] A. Classen ,Q. Boucher, P. Faber, and P. Heymans. « Syntax and Semantics of TVL, a Text-based Feature Modelling Language ». Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur,Belgium, 2010. (extended version of [21])

[21] A. Classen , Q. Boucher, P. Faber, and P. Heymans. « Introducing TVL, a text-based feature modelling language ». In Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS’10), Linz, Austria, January 27-29, pages 159–162. University of Duisburg-Essen, January 2010.

[22] R. Michel, A. Classen, A. Hubaux and Q. Boucher. « A Formal Semantics for Feature Cardinalities ». In Feature Diagrams Fifth International Workshop on Variability Modelling of Software-intensive Systems, Namur (Belgium), 27-29 January 2011.

[23] A. Hubaux. « Softare Product Lines Engineering and feature Modelling in a Nutshell ». PReCISE Research Center, University of Namur, Namur,Belgium, 2010

[24] P. Faber. « Conception d’un logiciel automatisant le contrôle et l’analyse de modèles TVL ». Faculté d’informatique, FUNDP, Namur, 2010.

[25] Marcelo Finger. « SAT Solvers : A brief Introduction ». Department of Computer Science, Universidade de Sao Paulo

[26] M. L. Griss, J. Favaro, M. d'Alessandro, “Integrating feature modeling with the RSEB”, Proceedings. Fifth International Conference on Software Reuse (Cat.No.98TB100203). IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, xiii+388 pp. p.76-85.

[27] B. Liskov, J. Guttag. « Program Development in Java : Abstraction, Specification, and Object-Oriented Design ». Addison-Wesley, 2001.

[28] OMG. « Object Constraint Language Version 2.2», OMG, 2010.

[29] « JFlex : The Fast Scanner Generator for Java ». Retrieved 2012-04-13 from http://jflex.de/

[30] « CUP : LALR Parser Generator for Java » Retrieved 2012-04-13 from http://www2.cs.tum.edu/projects/cup/

[31] Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In: Batory, D., Consel, C., Taha, W. (eds.): Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6–8, 2002, LNCS 2487, Springer-Verlag (2002) 156–172

[32] Deliverable D1.2, Full ABS Modeling Framework, Highly Adaptable and Trustworthy Software using Formal Models (HATS), EU FP7 Integrated Project, March 2011.

[33] D. Clarke. « Modelling Software Product Lines with the HATS Abstract Behavourial Modelling Language, November 2011.